UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

Honeywell Bull
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

NOTE DI SOFTWARE

ANNO 1987
36/37
GIUGNO-OTTOBRE

UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

Honeywell Bull
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

QUESTO NUMERO 1

— SECOND GENERATION EXPERT SYSTEMS,
di L. Steels 3

— AN IMPLEMENTATION OF THE KNUTH-BENDIX
ALGORITHM,
di M. P. Bonacina 9

— MC-OBJ: UN INTERPRETE IN "C" PER "OBJ",
di C. Cavenaghi, M. De Zanet, G. Mauri 16

— DOUBLE BIND AND EFFECTIVE COGNITIVE COUPLING
IN HUMAN - MACHINE INTERACTION,
di G. Tonfoni, F. Lanzara 27

— LINGUAGGIO I: UN SISTEMA PER LA RAPPRESENTAZIONE
DELLA CONOSCENZA IN LINGUAGGIO NATURALE,
di A. Fabris, M. Cimini 33

— GRAPHICS CODE GENERATOR BASED ON FORMAL
SEMANTIC DESCRIPTION PRINCIPLES,
di Ivan Jelinek 41

— ωLISP: UN AMBIENTE DI SVILUPPO PER A.I.
SU PERSONAL COMPUTER,
di E. Ciapessoni, F. Soresi 45

AVVENIMENTI

— REWRITING TECHNIQUES AND APPLICATIONS,
di M. P. Bonacina 52

— INTERNATIONAL JOINT CONFERENCE AN ARTIFICIAL
INTELLIGENCE,
di F. Gardin, A. Malacart 53

RECENSIONI
— a cura di H. Taylor 55

IL NOTIZIARIO SUI CD-ROM 56

CALL FOR PAPERS 58

NOTIZIARIO SISTEMI SOFTWARE

ANNO 1987
36/37
GIUGNO-OTTOBRE

QUESTO NUMERO

NOTE DI SOFTWARE si presenta, nel decimo anno dall'inizio delle pubblicazioni, con una veste editoriale rinnovata.

Da tempo la rivista ha abbandonato, per quanto riguarda le caratteristiche tipografiche, l'iniziale modello di appunti universitari o di laboratorio di ricerca, per orientarsi, anche grazie all'evoluzione della tecnologia editoriale, verso soluzioni che riteniamo più "formali" ed adeguate ad un pubblico più eterogeneo di lettori.

Con questo numero la rivista modifica la tradizionale copertina azzurra e sceglia un design più moderno che mantiene in grande evidenza la continuità della testata e accentua al tempo stesso l'impegno della proprietà e dell'editore.

HONEYWELL BULL ITALIA e UNIVERSITÀ DEGLI STUDI DI MILANO mediante il Centro Studi Informatica ed Automazione ed il Dipartimento di Scienze dell'Informazione proseguono nell'impegno di realizzazione e diffusione di uno strumento certamente unico di trasmissione di esperienze, metodologie ed informazioni nell'area della tecnologia del software.

La collaborazione alla pubblicazione resta come nel passato aperta ai contributi di Aziende, Enti e specialisti del settore.

Ed ora una breve presentazione dei contenuti di questo numero. Ampio spazio è ancora dedicato ad Expert Systems ed A.I. negli articoli di Luc Steels sull'acquisizione delle conoscenze dei sistemi esperti della seconda generazione e di E. Ciapessoni e F. Soresi sulla realizzazione di una stazione di lavoro per A.I. su personal computer.

All'argomento è dedicata anche una rubrica di Avvenimenti con la Conferenza Internazionale sull'Intelligenza artificiale tenutasi a Milano nello scorso agosto ed i cui aspetti salienti F. Gardin e A. Malacart hanno per noi sintetizzato e Recensioni curato per l'occasione da H. Taylor.

Il problema dell'interfaccia uomo-macchina è trattato negli articoli di M.P. Bonacina che descrive una implementazione dell'algoritmo di Knuth-Bendix, di G.F. Lanzara e G. Tonfoni che analizzano la problematica del "double bind", nella proposta del Linguaggio I di A. Fabris e M. Cimini e nella presentazione di M.P. Bonacina della conferenza internazionale di Bordeaux sulle tecniche di Rewriting per la Rubrica Avvenimenti.

Completa il panorama la presentazione dell'articolo di C. Cavenaghi, M. De Zanet e G. Mauri relativo ad un interprete scritto in linguaggio C per il linguaggio Obj e di un metodo per la costruzione di codici grafici presentato da I. Jelinek.

Ampia informazione sui CD-ROM nel notiziario dedicato a cura di A. Borgia, G. Dalto, G. Pagani, F. Vagliasindi.

Infine un Call for paper per la annuale nona conferenza Europea sulle Reti di Petri la cui teoria ed applicazione è stata sempre attentamente seguita da NOTE DI SOFTWARE.

La REDAZIONE

SECOND GENERATION EXPERT SYSTEMS

Luc Steels

Artificial Intelligence Laboratory

Vrjie Universiteit - Brussel

RIASSUNTO

I sistemi esperti della seconda generazione sono in grado di combinare il ragionamento eurustico basato sulle regole, con il ragionamento profondo basato su di un modello del dominio del problema. Questo risolve un numero dei principali problemi dei sistemi esperti esistenti, più importante tra tutti quello dell'acquisizione delle conoscenze: i sistemi esperti della seconda generazione sono in grado di imparare nuove regole esaminando i risultati del ragionamento profondo. Questo articolo presenta i componenti dei sistemi esperti della seconda generazione ed illustra un esempio.

ABSTRACT

Second generation expert systems are able to combine heuristic based on rules, with deep reasoning based on a model of the problem domain. This solves a number of major problems of current expert systems, most importantly the problem of knowledge acquisition: second generation expert systems can learn nex rules by examining the results of deep reasoning. The paper outlines the components of second generation expert systems and gives an example.

1. INTRODUCTION

In the last decade significant advances in the study of Artificial Intelligence have given rise to a new class of computer programs called expert systems. Expert systems are programs capable of expert level performance in specialised fields. They constitute an important development in computer usage because they make a whole new class of problems amenable to computational treatment.

The scientific, technological and business opportunities caused by this development are enormous. Many existing industrial and governmental organisations are now applying this new technology to their own aims, new businesses have sprung up, and expert systems have formed the subject of substantial national and international research programs all over the world.

What we see emerging today is a substantial jump in expert systems technology. The development is substantial because it touches on all aspects of an expert system: the nature of the potential problem domains, the type of reasoning being performed, the explanation capabilities, and most importantly the way knowledge acquisition is performed.

In brief, whereas first generation systems rely purely on heuristic knowledge in the form of rules, second generation systems have an additional component in the form of a deeper model which gives them an understanding of the complete search space over which the heuristics operate. This makes two forms of reasoning possible: (1) the application of heuristic rules in a style similar to the classical expert systems. (2) the generation and examination of a deeper search space following classical search techniques, if there are non heuristics.

A number of fundamental problems of current expert systems are solved using this extra complexity.

- First generation expert systems are brittle, in the sense that as soon as situations occur which fall outside the scope of the heuristics rules, they are unable to function at all. In such a situation, second generation system fall back on search which is not knowledge-driven and therefore

potentially very inefficient. However, because these traditional search techniques can theoretically solve a wider class of problems, there is a graceful degradation of performance instead of an abrupt failure.

• First generation systems base explanations purely on a backtrace of the heuristic rules that were needed to find a solution. It is well known however that the path followed to find a solution usually differs from a convincing rational argument why the solution is valuable, particularly if a lot of heuristic knowledge entered into the reasoning process. Because second generation systems have access to a deeper understanding of the search space, they are capable to formulate a deeper and more convincing explanation which goes beyond the mere recall of which rules fired.

• The most important advantage lies however in knowledge acquisition. Finding heuristic rules has turned out to be extremely difficult. Experts typically take a long time to come up with solid rules, the ruleset seems never complete, is continously changing and shows incosistencies across experts (and even within the same expert). These inconsistencies are apparently due to different experiences which are the source of heuristic rule discovery. Second generation expert systems constitute a major jump forward in current technology because they exhibit learning behavior in the sense that they are capable to acquire new heuristic rules.

The need for a new generation of expert systems has been felt for quite a while by various authors. An important milestone paper in this respect was [5] who discussed the strenghts and weaknesses of first generation expert systems. But evolutions in expert systems technology are by necessity slow because it takes at least five years to construct a serious system.

Second generation expert systems draw on a lot recent in A.I.. One source is the work on qualitative casual reasoning which provides ways to handle deep reasoning (see [1] for a recent overview). Another source is recent work on learning such as reported in the collection put together by [7]. Recent advances in the design and implementation of knowledge representation systems which have an open-ended set of formalisms and are capable of meta-representation and meta-reasoning are another enabling factor.

The objective of this paper is to outline the components of a second generation expert system and to illustrate briefly how each them works. The examples are drawn from one important application area, namely expert systems for repair and maintenance of technical system.

Technical systems, such as trains, computers, airplanes, power plants, or cars, are artefacts constructed to perform a particular function. In contrast to biological or natural systems which formed the domain of most first generation expert systems, they are in principle completely understood. All components are known and the behavior of the whole can theoretically be predicted from the behavior of the parts. Maintaining technical systems involves diagnosing possible failures (or the potential for future failures) and planning and executing a sequence of repairs that will lead again to a functioning system.

The repair and maintenance of technical systems is a skill that relatively few people posses. Skill is needed because technical systems are very complex. An exhaustive search of alla possible things that could go wrong is typically out of the question. Also, because not every component is observable, reasoning has to proceed on the basis of weak and very partial information.

The trouble-shooting of an ordinary car is taken here as specific application because many people have intuitions about it. Unfortunately, the exposition of the example (which have all been implemented) has to be brief, [9] contain more details.

2. COMPONENTS OF A SECOND GENERATION EXPERT SYSTEM

The kernel of an expert system consists of two components: a representational component and a problem solving component. Other components for communicating with the user, constructing explanations, gathering data, etc. are constructed around this kernel and will not be discussed in this paper.

THE REPRESENTATIONAL COMPONENT

The representational component has two subcomponents: a conceptual model and one or more factual models. The conceptual model delineates the basic entities in the universe of discourse, their possible properties and relations, as well as associated information like defaults, ways to compute certain properties, ways to communicate with the user, etc. The factual model contains all the facts, intermediate hypotheses and final conclusion for a specific problem instance using the conceptualisations provided by the conceptual model.

The representational component is typically frame-based. Information is structured in units with various slots that hold information about concept described by the unit. This information can be in the form of defaults, rules, procedures to compute information, etc. and is inherited by more specific units.

PROBLEM SOLVING COMPONENT

In first generation expert systems, the problem solving component consists solely of a collection of rules. A rule contains conditions (the IF-part) and inferences which can be drawn if the conditions are satisfied in the factual model under investigation (the THEN-part). Rules are typically structured in rule sets and activation is guided by a control structure which selects what rule to explore in case more than one rule is applicable. Recently there has been a trend to let the components of a rule be richer in structure, so that the various functions of the conditions or the conclusions become explicit. Thus a diagnostic rule

might have preconditions, typical symptoms, impossible symptoms, probable symptoms and necessary symptoms. See [4] or [3] for motivations and more examples. This finer structuring of a rule is also important for learning.

In second generation expert systems there is an additional problem solving component which performs deep reasoning. In comparison to the heuristic rules, this component explores a much larger search space because the primitive operators reflect a more fundamental and deeper description of the domain.

For many application domains a deep model is available. For example the anatomy and internal working of many technical devices is entirely known because it has been designed. Therefore it is possiblein principle to perform a thorough investigation, However such an investigation will be virtually impossible if a complex device is involved or if the cost of certain observations is very large. Hence a graceful performance degradation.

THE LEARNING COMPONENT

The discovery of heuristic rules as an ourcome of deep reasoning requires a third component, the learning component. It has two important subcomponents: one that extracts rules from deep reasoning (the rule-extractor) and one that incorporates the rules in the existing rule-set (the rule-integrator). Other learning operations can be performed on rule-sets to derive new concepts or optimise the rule-set.

TOOLS

There is a growing number pf commercially available tools to construct expert systems. These tools range from relatively simple expert system shells which support a particular rule formalism (such as the IF/THEN rules of OPS-5), to sophisticated knowledge engineering environments. Second generation expert systems require the more sophisticated class of tools to implement and integrate their various components.

First of all variety of formalisms needs to be supported: frame-structured representations to represent the models, rule formalisms to represent the heuristic knowledge, constraints to drive deep reasoning, etc. Moreover it must be possible to represent knowledge ABOUT each of these items, requiring the capacity for meta-representation and meta-reasoning. For example, it must not only be possible to represent and use heuristic rules, but also to represent facts or rules about the rules and reason explicitly over their structure. In addition, it must be possible to represent and reason explicitly over deductions that have been made, new concepts that have been introduced, or questions that have been asked.

Various knowledge representation systems now have the capability to perform these functions. In our own experiments we use the knowledge representatio system KRS [8].

KRS is designed to support the definition and manipulation of concepts. Systems based on KRS (and indeed KRS itself) consist of nothing but concepts. Concepts are structured in inheritance hierarchies and have a number of associated concepts (called subjects). There is a graphical interface to browse through networks of concepts and to interactively construct or edit them.

The construction of a specific application requires the introduction of domain concepts. But KRS also supports the definition and use of formal concepts. These are concepts about knowledge representation keeping information, such as a relation-formalism or a set-formalism. There are computational concepts to describe algorithms, and concepts defining knowledge representation formalisms like a rule, a logical implication, or a constraint. Control-structures and inheritence schemata are also described explicitly as concepts.

3. THE PROBLEM SOLVING COMPONENT

3.1 Ther task

A casual model consists of a set of properties of components which are causally related in the sense that the value of one property is determined by the values of one or more other properties. Some of these properties are observable, most of them are not or only with difficulty. Causal relations can easly be represented in a network where the nodes represent the properties of the components.

There are three requirements for an engine start. The starter must turn the engine, the two sparkplugs must fire and the starter-transmission must be okay. Each of these has a number of further enabling causes.

The following task will be taken as example application: given a technical device described as a network of causal relations between properties of components, given a state of the device and an anomalous property, i.e. a property whose observed value differs from the expected value, find an explanation either in terms of mafunctioning components or in terms of external controls that need to be changesd.

The task assumes a human, (or computer) capable of observing properties and judging whether the property is supposed to be the way should be based on a fuctional model.

3.2 Heuristic rules

It is easy to envisage heuristic rules for this task. For example, there might be rules like "if the headlights are not on, check whether they have been turned on", or "if the lights cannot burn, the batteries are not charged". Most user's manuals contain such rules and even non-experts know at least some of them.

3.3 Deep reasoning

Alternatively it is possible to look at the detailed casual network of the car. We will give one example but other deep reasoning mechanisms would work equally well. Deep reasoning can be done using the following principles:

1. If the effect of a property is normal, then there is reason to believe that it is itself normal.

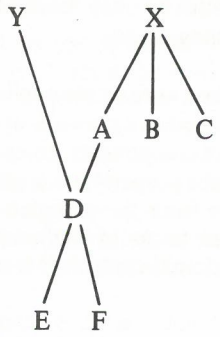
2. If the effect of a property is anomalous, it may itself be anomalous too.

Thus if the engine is not starting and one of its causes is the presence of gas, then this property becomes suspect.

A possible deep reasoning algorithm starts from an anomalous property and investigates the causes of this property until observable properties are arrived at. The user is queried for these proporties and if they are anomalous (i.e. different from his expectation), an explanation has been found. If they are normal (i.e. matching with the expectation) then the explanationn for the original anomaly lies elsewhere.

When none of the observed properties which ultimately caused the anomalous property were anomalous, the cause of the anomaly must lie somewhere in between. In other words, one of the internal components must be faulty.

The number of faulty components can be reduced if there are other observables who were caused by the same properties. Consider for example, the following network which depicts rhe causes emanating from X, and Y.



X is anomalous and needs to be explained. E, F, B, and C are observable properties and they are all normal. So the explanation why X is anomalous must be because there is a faulty component in between. But if Y is also normal, it can be deduced that its cause, namely D must also be normal, and therefore the only possible explanation for the anomaly of X is that A is anomalous, i.e. the component of which A is a property must be malfunctioning.

Let us consider an example where the network given earlier is examined because the starter is not turning. The other relevant observations are:

Starter = Turning is Anomalous
Battery = Charged is Normal
Contact = On is Anomalous

Backward examination of the causes of Starter=Turning leads to Starter=Powered. This is not observable so reasoner descends further in the network. The causes of Starter=Powered are Battery=Charged and Contact=On. Battery=Charghed is normal so that cannot provide the explanation for the original anomaly. On the other hand. Contact=On is anomalous, therefore an explanation has been found.

3.4 Differences between heuristic rules and deep reasoning

Based on the above discussion the differences between heuristic and deep causal reasoning become clearer. Thus, there could be a heuristic rule that says "when the headlights are not on, check wherther they have turned on". A deep reasoning system would follow the causal connections starting from the headlights and would eventuallly arrive at the switch to turn them on, but it would possibly check many other things on the way or start wandering around in the causal network in directions that have little to do with the initial problem.

So we see the following differences:

heuristic rules make shortcuts: intermediate steps are skipped because those steps are unlikely to lead to a quick solution of the problem;

heuristic reasoning does not refer to the working of the internals of the system but associates a number of easy to make external observations with a plausible conclusion.

A deeper causal model will be able to come to the same conclusion, but would in principle also examine many other things. Some of which might not be easy to answer without extra measuring apparatus.

On the other hand the set of heuristic rules is necessarily incmplete because it wuold require a rule for every possible failure in the system and thus power (in terms of efficiency in problem solving) obtained from heuristic rules would be lost.

What we want of course is the best both: use heuristic rules for common cases to get a quick solution to the problem but fall back on deep reasoning when the case is not covered by heuristic rules. This is precisely what a second generation expert system does. It makes it also possible to extract heuristic rules after a deep reasoning sequence has been done. The next section shows how.

4. THE LEARNING COMPONENT

A deep reasoning sequence potentially investigates many causal connections and queries the user for lot of observations. To be worthwhile, heuristic rules must do the oppo-

site: they should be as simple as possible, leaving out all the details and all the observations that did not directly contribute to a solution of the problem. Extracting rules from deep reasoning is difficult becuase it is not obvious which details are irrilevat. Steels and Vand De Velde in [9] describe a technique, called rule-learning by PROGRESSIVE REFINEMENT, which adresses this issue.

Initially a new rule makes as much abstraction as possible using circumscriptive reasoning: everything which is not mentioned in a rule supposed to be irrilevant or normal. Later the conditions of a rule are progressively refined to discriminate with a new rule whose conditions of a rule are progressively defined to discriminate with a new rule whose conditions overlap but whose conclusions are different. Interestingly enough, this technique is monotonic. A learned rule never becomes invalid, although its applicability is restricted.

We consider learning a particular type of rules (called solution rules) which heve primary symptoms, secondary symptoms and corrected properties. The rule becomes active when all its primary symptoms are present. Then the secondary symptoms are checked and when they are present as well, a number of properties are to be corrected. (Other construction procedures are needed for other types of rules although the principles remain the same).

A rule can be extracted from a deep reasoning sequence as follows:

1. The primary-symptoms of the rule are equal to the anomalous property that triggered the deep reasoning sequence.
2. The corrected-properties are equal to the conclusions of the deep reasoning sequence, i.e. the list of properties which deep reasoning deemed ultimately responsible for the anomaly. It is assumed that a canghe of these properties either directly or indirectly causes the anomaly to disappear.
3. The secondary-symptoms are initially ampry.

For example, after the first deep reasoning sequence discussed in the previous section, the following rule can be extracted:

Rule-1
  Primary-symptoms: Not Starter=turning
  Secondary-symptoms: -
  Corrected-properties: Contac=on

Cleary this is an extreme form abstraction. It assumes that the encountered situation is the only situation that can ever go wrong! The triggering symptom is therefore linked directly with the correction that worked.

New rules need to be integrated with already existing rules to make sure that they remain mutually exclusive, i.e. no two rules should fire and propose a different solution for the same primary-symptom.

Rule-integration is trivial if the rules are orthogonal, otherwise the following procedure is used:

1. The corrected-properties of the new rule are added as necessary symptoms of the old rules, unless the property is one of the corrected-properties of the old rule.
2. The negation of the corrected-propertis of the new rule is added as a secondary symptom of the new rule.

The intuition behind this integration procedure is that symptoms are added from other rules to make sure that the rules remain mutually exclusive. The refinement process is monotonic and works also for rule chaining.

For example, suppose we have a rule which says that the car should be put in parking mode when does not start:

RULE-2
  primary-symptoms: Not car=starts
  secondary-symptoms: -
  corrected-properties: mode=parking

Suppose there is now a new which says that there should be gas put in the car when the car does not start:

RULE-3
  primary-symptoms: Not car=Starts
  secondary-symptoms: -
  corrected-properties: gas=present

then the ruleset after intergration looks as follows:

RULE-2
  primary-symptoms: not Car=Starts
  secondary-symptoms: gas=present
  corrected-properties: mode=parking

RULE-3
  primary-symptoms: not car=starts
  secondary-symptoms: not gas=present
  corrected-properties: gas=present

The rules now say: if the car does not start and there is gas present, put the car in parking mode. If the does not start and there is no gas present, make gas be present.

If a new problem is posed to the system, the rules constructed followig the above procedure are tried before any deep reasoning is done. If a rule fires because the primary-symptom and the secondary-symptoms have been observed, then the corrected-properties are tested out in the model and it is investigated whether their cerrection indeed resolves the initial anomaly. If this is the case then the rule is confirmed and no changes need to be made. If not, deep reasoning is started again to perform a more thorough diagnosis. From the result a new rule can then be extracted and incorporated.

FURTHER LEARNING

Although the ruleset obtained through the above construction process will progressively solve more and more problems, it is far from efficient. Rules that partially overlap remain independent, the same conditions will be

checked many times over, etc. Therefore other construction procedures are introduced that operate once the initial ruleset have been formed. Here are some example:

- *LEARNING THROUGH RULE COMBINATION*

Rules that have structure in common can be combined. For example, if two rules have the same set of corrected-properties, then a new rule can be generated with a combination of primary symptoms and secondary symptoms.

- *LEARNING THROUGH GENERALISATION*

Deep reasoning typically takes place over a small part of a complex system. Often there are other portions of the same device (or different devices) which have the same causal structure. By making abstraction of the specific area for which it was discovered, rules get a wider applicability. If the causal structures are not equal but similar, then further learning through adaptation can refine the ruleset acquired elsewhere.

- *LEARNING NEW CONCEPTS*

A subset of the necessary symptoms may be common to many rules. In such cases it is possible to introduce a new concept which can be used to formulate an intermediate conclusion. It will be more efficient to first establish this intermediate conclusion, instead of reconsidering all of its components. The new concept can also be used to enrich the model of the device.

- *LEARNING THROUGH RULE-ORDEING*

The rules needs to be restructured so that rules are ordered. This leads to greater efficiency because symptoms present in more than one rule are not investigated more than once. The ordening itself again depends on experince, so that rules covering situations which are more frequent get priority.

- *GROUPING OF RULES*

Another important form of learning is the division of rules in smaller sets and the learning of rules that lead from one set to another set. This division can often be based on the underlying model.

5. CONCLUSIONS

A second generation expert system not only uses heuristic rules, but has also a model of the domain so that deeper reasoning is possible if the rules are inadequate. This leads to a graceful performance degradation instead of an abrupt failure if there is no rule covering a particular situation. It also leads to a new way of doing knowledge acquisition by extracting rules from experience in solving a particular problem.

The state of the art in Artificial Intelligence is sufficiently advanced to realise second generation expert systems. Our own experiments in the domain of diagnosis of technical systems have confirmed the feasibility.

6. ACKNOWLEDGEMENT

Many people atthe VUB AI lab. and particularly Walter Van De Velde, have in one way or another contributed to the ideas presented in this paper or to their implementation. The KRS-system is a team effort with Peter Van Damme and Kris Van Marcke as prime implementers of the LISP-machine version used for the experiments reported here. Leo De Wael and Patrick Backx are other members of the team involved in technical expert systems. They have been instrumental in the development of models for deep and heuristic reasoning in this domain. Van De Velde has developed and implemented the learning experiments. This research is partially sponsored by the Belgian Ministry of Scientific Planning.

7. REFERENCES

[1] D. Bobrow, P. Hayes, SPECIAL ISSUES ON QUALITATIVE REASONING, in Artificial Intelligence Journal, North-Holland Pub. Co., 1984

[2] D. Bobrow, T. Winograd, KRL. A KNOWLEDGE REPRESENTATION LANGUAGE, Cognitive Science, Vol. 1, 1. Lawrence Erlbaum, New Haven, 1977

[3] J. Breuker, B. Wielinga, KADS: STRUCTURED KNOWLEDGE ACQUISITION FOR EXPERT SYSTEMS, in Proceedings of Fifth International Workshop on Expert Systems, section 10a, Avignon, France, 1985

[4] W. Clancey, THE EPISTEMOLOGY OF RULE-BASED SYSTEMS, Artificial Intelligence Journal, North-Holland Pub. Co. Amsterdam, 1982

[5] R. Davis, EXPERT SYSTEMS: WHERE ARE WE AND WHERE ARE WE GOING, The Al Magazine, Winter, 1983

[6] R. Greiner, D. Lenat, RLL. A REPRESENTATION LANGUAGE LANGUAGE, Proceedings Of AAAi-82, Kaufman, Los Angeles, 1982

[7] R. Michalski, J. Carbonell, T. Mitchell, MACHINE LEARNING, AN ARTIFICIAL INTELLIGENCE APPROACH, Springer-Verlag, Berlin, 1984

[8] L. Steels, THE OBJECT-ORIENTED KNOWLEDGE REPRESENTATION SYSTEM KRS, In T. O'Shea (ed.) Proceedings of ECAI-84, North-Holland Pub. Amsterdam, 1984

[9] L. Steels, W. Van De Velde, LEARNING IN SECOND GENERATION EXPERT SYSTEMS, Chapter 10 In Kowalik (ed.), Knowledge-based Problem Solving, Prentice-Hall, Englewood Cliffs, New Jersey, 1985

AN IMPLEMENTATION OF THE KNUTH-BENDIX ALGORITHM

Maria Paola Bonacina

Dipartimento di Scienze dell'Informazione

Università di Milano

RIASSUNTO

Questo articolo descrive un'implementazione dell'algoritmo di Knuth-Bendix in linguaggio C su Macintosh, realizzata dall'autore presso il Dipartimento di Scienze dell'Informazione. I principali concetti teorici relativi all'algoritmo sono brevemente ricordati attraverso la descrizione dell'implementazione, che cerca di mantenere il punto di vista dell'utente. È incluso un ricco insieme di esempi di esecuzioni. La bibliografia dà suggerimenti a chi voglia conoscere meglio l'algoritmo di Knuth-Bendix.

ABSTRACT

This is a report of an implementation of the Knuth-Bendix algorithm realized by the author in C language on a Macintosh at Dipartimento di Scienze dell'Informazione. The main concepts underlying the algorithm are briefly sketched through the description of the implementation, oriented to give the user a feeling of what he or she can do with the program. For this purpose a rich set of examples of runs is given. Finally the references provides suggestions for the reader interested in knowing more about the algorithm.

1. INTRODUCTION

This work is not intended to explain Knuth-Bendix algorithm: for this purpose the reader is referred to the references [1, 3, 4, 5, 6, 15, 16, 23]. This paper is rather intended to give account of the author's personal experience in implementing and testing the Knuth-Bendix algorithm.

The following presentation of the Knuth-Bendix program tries to meet a naïve user needs: implementation details are omitted to explain instead what a user can do interacting with this program.

The aim of this implementation of the Knuth-Bendix algorithm is to provide a tool for collecting experiences about the algorithm's behaviour. The algorithm is conceived as a rather low level inference engine: it should be the "hardware" of a virtual machine working on term rewriting systems. For this reason and for efficiency's sake the C programming language [21] has been chosen as the implementation language.

Three main functionalities are available: a canonical simplifier [5, 6], the original Knuth-Bendix algorithm [3, 4, 5, 6, 15, 16, 23] and the Unfailing Knuth-Bendix algorithm [5, 14].

2. GETTING STARTED

The first step is to introduce a set of axioms obeying the following grammar:

1) <axiom> ::= <term> <separator> <term>

2) <separator> ::= = | →

3) <term> ::= <var> | <const> | <ident>(<arglist>)

4) <var> ::= <identvar>

5) <const> ::= <ident>

6) <arglist> ::= <term> | <term>,<arglist>

7) <ident> ::= <lowerletter> | <lowerletter> <seqchar>

8) <identvar> ::= <upperletter> | <upperletter> <seqchar>

9) <seqchar>::=<letter> | <digit> | <letter> <seqchar> | <digit> <seqchar>

10) <lowerletter> ::= a|...|z

11) <upperletter> ::= A|...|Z

12) <digit> ::= 0|...|9

The first two productions show that both unoriented equations and oriented rewrite rules are accepted.

The third and fifth productions show that terms are written in prefix notation and only identifiers are allowed as constants: prefix notation is chosen as the one usually adopted for terms in first order languages; numerical constants are forbidden since they could suggest that natural numbers are built in: no model is built in, all that the algorithm knows is the given signature.

An input file can contain comments:

13) <comment> ::= # <seqchar> #

When the system reads the equational theory from a file it looks for a table whose entries obey two more productions:

14) <tableentry>::=<ident>|<number>|<number>

15) <number> ::= <digit> | <digit> <number>

An entry specifying an index and a weight must be given for every function or constant symbol.

If the system reads the equational theory from standard input it asks interactively for indexes and weights, which are necessary to compute the Knuth-Bendix reduction ordering on terms [4, 23]. There are many other reduction orderings defined in the literature [1, 7, 9, 17, 19, 26]; the Knuth-Bendix ordering has been chosen for historical reasons, since it was born with the algorithm itself.

A top-down parser analyses the theory and translates it in internal representation: terms are represented as binary trees with structure sharing on variables.

First of all equations are oriented as rewrite rules following the Knuth-Bendix ordering on terms.

Now the user can choose between Reduce, i.e. the canonical simplifier, KB, i.e. the original Knuth-Bendix algorithm, and UKB, i.e. the Unfailing Knuth-Bendix algorithm.

3. THE CANONICAL SIMPLIFIER

A canonical simplifier reduces a term to normal form with respect to a given term rewriting system. The implemented canonical simplifier is called by the user, by KB or by UKB.

The user can introduce terms to be simplified just after the equational theory and select the command Reduce to get them into normal form.

KB and UKB call the canonical simplifier to keep the current term rewriting system interreduced: every term in the data base is reduced as much as possible.

The following factors are important for a canonical simplifier:

— terms are represented as trees

— the data base of rules is dynamically changing

— the term rewriting system is noetherian [25].

This simplifier scans trees in preorder, implementing outermost-innermost left-right rewriting.

The term rewriting system is noetherian since it is included in a noetherian ordering, the Knuth-Bendix reduction ordering: the canonical simplifier computes a total function, i.e. its computations stop independently of the chosen rewriting strategy. This property is called strong normalizability [2, 22].

A rather high portion of Knuth-Bendix algorithm execution time is spent in rewriting terms, since every change in the data base imposes a recomputation of normal forms.

4. THE KNUTH-BENDIX ALGORITHM

The algorithm infers new rewrite rules from the given ones using the superposition technique.

This process can run to completion or stop with failure or stop with overflow: a failure signals that an equation is unorderable in the available reduction ordering, while an overflow signal means a probably infinite set of rewrite rules should be generated [3, 4, 5, 15, 16, 23].

Failures can be avoided by selecting the option of the Unfailing Knuth-Bendix algorithm, which keeps running even when there are unorderable pairs of terms. It requires slight modifications to the canonical simplifier and the superposition algorithm when unoriented equations are involved; a subsumption check is also necessary for UKB to avoid infinite generation of instances of the same pattern [5, 24].

After the option KB or UKB has been selected, three degrees of interactivity are available:

1. the user chooses the axioms selection strategy and then the computation runs on its own

2. the user controls the strategy manually; the execution is suspended every time a successor set has been generated and the user is given the possibility to resume or stop

3. like the preceding one; the user can also change strategy when the computation is suspended.

This opportunity is very interesting because it allows the user to drive the computation.

The axioms selection strategy determines the order in which axioms are selected for superposition. Axioms are identified by a history number: the input rules are numbered in the order they are read; when a new rule is generated is gets the successor of the highest number currently present. The following strategies are available:

1. first in — first out (FIFO)
2. last in — last out (LIFO)
3. smallest components and FIFO to decide conflicts
4. smallest components and LIFO to decide conflicts
5. by size and FIFO to decide conflicts
6. by size and LIFO to decide conflicts
7. ordered axioms with identity
8. ordered axioms with identity up to variables renaming.

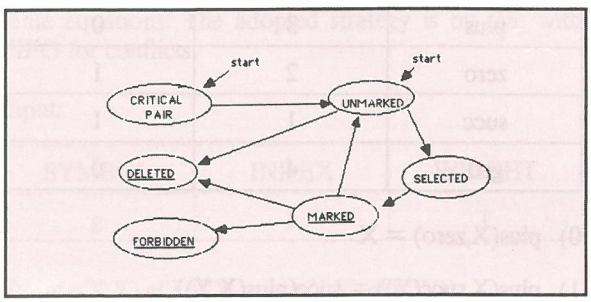
Let's define the weight of a term or tree as its number of nodes: smallest components strategy [24] chooses the rule such that the maximum between its left hand side's weight and its right hand side's weight is the lowest; when two rules share this same value another strategy is needed to solve the conflict.

By size strategy chooses the rule such that its left hand side's weight is the lowest; as before conflicts can occur.

Ordered axioms strategy uses an ordering on rules induced by the underlying reduction ordering on terms [19]: $\lambda \rightarrow \rho < \alpha \rightarrow \beta$ if and only if $\lambda < \alpha \vee (\lambda = \alpha \wedge \rho < \beta)$, where $=$ is syntactic identity in the seventh strategy, while it is identity up to variables renaming in the eighth.

FIFO, smallest components and by size are complete strategies, LIFO is not complete [24].

The following state transitions diagram shows a rule's history during Knuth-Bendix execution:



An input rule starts in *unmarked* state: an *unmarked* rule has not yet been used for superposition; whereas a *marked* rule has already been used.

An inferred rule starts in *critical — pair* state and later it becomes *unmarked.*

An *unmarked* rule becomes *selected* when the strategy chooses it and later it switches to marked state.

A *marked* rule gets back to *unmarked* state when its left hand side is modified.

A *marked* or *unmarked* rule is *deleted* when both sides reduce to the same term.

A *forbidden* rule has the form $c \rightarrow d$, where c and d are constants, or $f(x_1,...,x_n) \rightarrow h(y_1,...,y_n)$, where f and h are function symbols and x_i and y_i are variable symbols: these axioms mean the constant c and the function f are substituted throughout by d and h respectively; after these rules are applied to let c and f disappear they are useless for superposition and reduction: being in *forbidden* state they are skipped by the algorithm.

5. RUNNING KNUTH-BENDIX ALGORITHM

This paragraph reports some executions of the implementation of Knuth-Bendix algorithm on a Macintosh. The selected set of examples does not pretend to be exhaustive: it is rather intended to give the reader a flavour of what a Knuth-Bendix run looks like. There are examples for every possible outcomes of KB and UKB.

In the following every rule is preceded by its history number.

The first example shows completion of (r,l)system [23] by the Knuth-Bendix algorithm. The strategy is by size with FIFO to solve conflicts.

Input:

SYMBOL	INDEX	WEIGHT
f	1	0
e	2	1
i	3	0

0) f(X,e) = X

1) f(i(X),X) = e

2) f(f(X,Y),Z) = f(X,f(Y,Z))

Total number of axioms: 3.

Output:

0) f(X,e) → X

1) f(i(X),X) → e

2) f(f(X,Y),Z) → f(X,f(Y,Z))

3) i(e) → e

8) f(X,i(i(Z))) → f(X,Z)

14) f(i(Y),f(Y,Z)) → i(i(Z))

7) f(e,Z) → i(i(Z))

13) f(X,f(i(i(Z0)),Z)) → f(X,f(Z0,Z))

20) f(X,f(Z0,i(Z0))) → X

25) f(X,f(Z0,f(i(Z0),Z))) → f(X,Z)

26) i(i(i(Z0))) → i(Z0)

43) i(f(Y,Z)) → f(i(Z),i(Y))

Total number of axioms: 12.

The next example is taken from [10]: it shows a nasty case, where just one rule causes nontermination superposing onto itself.

Input:

SYMBOL	INDEX	WEIGHT
g	1	1
f	2	1

0) g(f(g(X))) = f(g(X))

Total number of axioms: 1.

Output:

0) g(f(g(X))) → f(g(X))

1) g(f(f(g(X)))) → f(f(g(X)))

2) g(f(f(f(g(X))))) → f(f(f(g(X))))

3) g(f(f(f(f(g(X)))))) → f(f(f(f(g(X)))))

4) g(f(f(f(f(f(g(X))))))) → f(f(f(f(f(g(X))))))

Total number of axioms: 5.

Let's now look at an example taken from [10], where Knuth-Bendix algorithm fails. This output has been obtained with the following strategies: by size with FIFO for conflicts, FIFO, LIFO, smallest components with FIFO for conflicts.

Input:

SYMBOL	INDEX	WEIGHT
plus	3	0
zero	2	1
succ	1	1
gcd	4	0

0) plus(X,zero) = X

1) plus(X,succ(Y)) = succ(plus(X,Y))

2) gcd(X,zero) = X

3) gcd(zero,X) = X

4) gcd(plus(X,Y),X) = gcd(X,Y)

Total number of axioms: 5.

Output:

0) plus(X,zero) → X

1) plus(X,succ(Y)) → succ(plus(X,Y))

2) gcd(X,zero) → X

3) gcd(zero,X) → X

4) gcd(plus(X,Y),X) → gcd(X,Y)

5) gcd(X,X) → X

6) gcd(succ(plus(X,Y)),X) → gcd(X,succ(Y))

7) plus(zero,Y) → Y

8) gcd(X,succ(zero)) = gcd(succ(X),X): failure

Total number of axioms: 9.

Running UKB on this input instead than KB the failure problem is bypassed but a neverending computation results.

Input:

SYMBOL	INDEX	WEIGHT
plus	3	0
zero	2	1
succ	1	1
gcd	4	0

0) plus(X,zero) = X

1) plus(X,succ(Y)) = succ(plus(X,Y))

2) gcd(X,zero) = X

3) gcd(zero,X) = X

4) gcd(plus(X,Y),X) = gcd(X,Y)

Total number of axioms: 5.

Output:

0) plus(X,zero) → X

1) plus(X,succ(Y)) → succ(plus(X,Y))

2) gcd(X,zero) → X

3) gcd(zero,X) → X

4) gcd(plus(X,Y),X) → gcd(X,Y)

5) gcd(X,X) → X

6) gcd(succ(plus(X,Y)),X) → gcd(X,succ(Y))

7) plus(zero,Y) → Y

8) gcd(succ(X),X) = gcd(X,succ(zero))

9) gcd(succ(succ(plus(X,Y))),X) → gcd(X,succ(succ(Y)))

10) gcd(succ(succ(X)),X) = gcd(X,succ(succ(zero)))

11) gcd(succ(succ(succ(plus(X,Y)))),X) → gcd(X,succ (succ(succ(Y))))

12) gcd(succ(succ(succ(X))),X) = gcd(X,succ(succ (succ(zero))))

Total number of axioms: 13.

Two infinite sequences of axioms are generated: gcd(succ^n (X),X) = gcd(X,succ^n(zero)) and gcd(succ^n(plus(X,Y)),X) → gcd(X,succ^n(Y)). The first one is caused by axioms 0 superposed on 6, 9, 11, ... , i.e. the elements of the second sequence, which is caused by axioms 1 superposed on 4, 6, 9, 11 ...

The following term rewriting system, taken from [14], present a structure called entropic groupoid. Its completion by UKB shows the subsumption check is necessary: without it the program produced many instances of the same equations. The adopted strategy is by size with FIFO for conflicts.

Input:

SYMBOL	INDEX	WEIGHT
g	1	1

0) g(g(X,Y),g(Z,W)) = g(g(X,Y),g(Y,W))

1) g(g(X,Y),X) = X

Total number of axioms: 2.

Output:

4) g(g(X,Y),W) = g(g(X,Z),W)

1) g(g(X,Y),X) → X

2) g(Z,g(Y,W)) → g(Z,W)

7) g(g(g(X,Z),Y),Y0) → g(X,Y0)

Total number of axioms: 4.

The next example shows the strong impact of strategy choice on Knuth-Bendix algorithm work: in the first run, by size strategy with FIFO for conflicts gives failure on the unorderable equation 43; switching to by size strategy with LIFO for conflicts the computation ends successfully. This term rewriting system [23] presents groups with two identities and two inverses: KB proves instead they are unique.

Input:

SYMBOL	INDEX	WEIGHT
f	2	0
e	1	1
i	5	0
u	3	1
j	4	1

0) f(e,X) = X

1) f(i(X),X) = e

2) f(f(X,Y),Z) = f(X,f(Y,Z))

3) f(u,X) = X

4) f(j(X),X) = u

Total number of axioms: 5.

Output:

0) f(e,X) → X

1) f(i(X),X) → e

2) f(f(X,Y),Z) → f(X,f(Y,Z))

18) i(e) → e

38) f(X,f(Y,i(f(X,Y)))) → e

40) f(X,f(Y,f(i(f(X,Y)),Z))) → Z

6) f(i(Y),f(Y,Z)) → Z

33) i(i(Y)) → Y

41) f(X,f(Y1,f(Y,i(f(X,f(Y1,Y)))))) → e

34) f(i(X,Y)),f(X,f(Y,Z))) → Z

42) f(Y,i(f(Y,Y))) → i(Y)

43) f(i(f(X,Y)),X) = f(Y,i(f(Y,Y))): failure

23) f(Z,e) → Z

14) u → e

28) f(Z,i(Z)) → e

29) f(Z1,f(i(Z1),Z)) → Z

30) j(Z) → i(Z)

Total number of axioms: 17.

Second output:

0) f(e,X) → X

1) f(i(X),X) → e

2) f(f(X,Y),Z) → f(X,f(Y,Z))

36) f(Y,f(i(Y),Z)) → Z

46) i(f(Y,Z)) → f(i(Z),i(Y))

6) f(i(Y),f(Y,Z)) → Z

19) u → e

16) f(Z,e) → Z

28) i(i(Y)) → Y

30) f(Y,i(Y)) → e

31) j(Y) → i(Y)

17) i(e) → e

Total number of axioms: 12.

Let's look how KB solves an exercise from [22]: it is about proving f(X) = X from f(f(f(X))) = X and f(f(f(f(f(X))))) = X, which are at once reduced to axioms 0 and 1.

Input:

SYMBOL	INDEX	WEIGHT
f	1	1

0) f(f(f(X))) = X

1) f(f(X)) = X

Total number of axioms: 2.

Output:

2) f(X) → X

Total number of axioms: 1.

6. CONCLUSIONS

A comparison between rewriting based systems is out of the scope of this paper: in [8, 9, 20] the interested reader can find complete presentations of three well known rewriting based systems whose core is the Knuth-Bendix procedure.

A great amount of work is still to be done to get a full laboratory on term rewriting systems: the first step should be the implementation of an AC-unification algorithm and of other reduction orderings. Since the main aim of this work is theorem proving, two concurrent directions are going to be pursued: an implementation of methods proposed in [17, 18, 19] and a study of the general problem consisting in expressing a query as a set of rewrite rules to manage it with Knuth-Bendix algorithm.

The overall result of this experience is by now the following: some amount of knowledge is required to work with Knuth-Bendix algorithm, for instance knowledge about the implemented reduction ordering. This means there is a level of metaknowledge above the Knuth-Bendix level: a real great improvement would be to automate this metaknowledge.

7. ACKNOWLEDGEMENTS

I'd like to thank Giovanni Degli Antoni for fruitful discussions and useful comments, Filippo Soresi for his help in Macintosh style interface implementation, Jeremy Dick and the referee for nice remarks.

8. REFERENCES

[1] L. Bachmair, N. Dershowitz, J. Hsiang, ORDERINGS FOR EQUATIONAL PROOFS, Proc. IEEE Symp. Logic in Computer Science, Cambridge, Massachussets, pp. 346-357, 1986

[2] C. Böhm, M. Dezani Ciancaglini, TO WHAT EXTENT CAN OR MUST COMPUTATIONS BE PARALLELIZED, in New Concepts and Technologies in Parallel Information Processing, ed. by E.R. Caianiello, Nato Series, 1975

[3] M.P. Bonacina, PRESENTAZIONE DELL'AL-

GORITMO DI KNUTH-BENDIX, Internal Report no. 7/86, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 1986

[4] M.P. Bonacina, L'ALGORITMO DI KNUTH-BENDIX, Tesi di laurea in Scienze dell'Informazione, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 1986

[5] M.P. Bonacina, DATA-FLOW KNUTH-BENDIX ALGORITHM AS INFERENTIAL ENGINE ON TERM REWRITING SYSTEMS, Internal Report no. 1/87, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, 1987

[6] B. Buchberger, BASIC FEATURES OF THE CRITICAL - PAIR/COMPLETION PROCEDURE, Invited lacture, Proc. of the 1st Conf. on Rewriting Techniques and Applications, J.P. Jouannaud (ed.), pp. 1-45, LNCS 202 Springer Verlag, 1985

[7] N. Dershowitz, ORDERINGS FOR TERM REWRITING SYSTEMS, Theoretical Computer Sciences 17, pp. 279-301, North Holland Publishing Company, 1982

[8] A.J.J. Dick, ERIL-EQUATIONAL REASONING: AN INTERACTIVE LABORATORY, Rutherford Appleton Laboratory, Report RAL-86-010, 1985

[9] R. Forgaard, A PROGRAM FOR GENERATING AND ANALYZING TERM REWRITING SYSTEMS, Master Thesis, MIT, 1984

[10] M. Hermann, I. Privara, ON NON TERMINATION OF KNUTH-BENDIX ALGORITHM, Research Report of the Institute of Socio-Economic Information and Automation in Management - Department of Programming System VUSEI-AR-OPS-3/85, 1985 or Proc. on 13th Colloquium on Automata, Languages and Programming, LNCS Springer Verlag, pp. 146-156, 1986

[11] J. Hsiang, TWO RESULTS IN TERM REWRITING THEOREM PROVING, Proc. of 1st International Conf. on Rewrite Techniques and Applications, 1985

[12] J. Hsiang, REFUTATIONAL THEOREM PROVING USING TERM REWRITING SYSTEMS, Artificial Intelligence 25, 255-300, 1985

[13] J. Hsiang, REWRITE METHOD FOR THEOREM PROVING IN FIRST ORDER THEORY WITH EQUALITY, Dept. of Computer Science, State University of New York at Stony Brook, Stony Brook, NY, 11794

[14] J. Hsiang, M. Rusinowitch, ON WORD PROBLEMS IN EQUATIONAL THEORIES, To appear in the proceedings of the 14th ICALP, 1987

[15] G. Huet, CONFLUENT REDUCTIONS: ABSTRACT PROPERTIES AND APPLICATIONS TO TERM REWRITING SYSTEMS, J.ACM 27, 797-821, 1980

[16] G. Huet, A COMPLETE PROOF OF CORRENCTNESS OF THE KNUTH-BENDIX COMPLETION ALGORITHM, J. Comput. and Syst. Sciences (USA) vol. 23, no. 1, pp. 11-21, 1981

[17] G. Huet, D.C. Oppen, EQUATIONS AND REWRITE RULES: A SURVEY, in R. Book (ed.) Formal Language Theory: Perspectives and Open Problems, Academic Press, New York, pp. 349-405, 1980

[18] J.M. Hullot, A CATALOGUE OF CANONICAL TERM REWRITING SYSTEMS, Technical Report CSL-113, SRI International, 1980

[19] D. Kapur, P. Narendran, THE KNUTH-BENDIX COMPLETION PROCEDURE AND THUE SYSTEMS, SIAM J. Comput. (USA), vol. 14, no. 4, 1052-1072, 1985

[20] D. Kapur, G. Sivakumar, ARCHITECTURE OF AND EXPERIMENTS WITH RRL, A REWRITE RULE LABORATORY, Proc. of a NSF Workshop on the Rewrite Rule Laboratory, Sept. 6-9, 1983 General Electric CRD Report 84GEN004, pp. 33-56, 1984

[21] B. Kernigham, D. Ritchie, THE C PROGRAMMING LANGUAGE, Prentice Hall, 1978

[22] J.W. Klop, TERM REWRITING SYSTEMS, Proceedings of First Autumm Workshop on Reduction Machines, Ustica, Italy, 1985

[23] D.E. Knuth, P. Bendix, SIMPLE WORD PROBLEMS IN UNIVERSAL ALGEBRAS, Proc. of the Conf. on Computational Problems in Abstract Algebra, Oxford, 1967, (ed. by J. Leech) Pergamon Press, Oxford, 263-298, 1970

[24] W. Küchlin, A THEOREM PROVING APPROACH TO THE KNUTH-BENDIX COMPLETION ALGORITHM, Computer algebra EUROCAM 82 (ed. by J. Calmet) in Lecture Notes in Computer Science (ed. by G. Hartmanis) Springer Verlag

[25] W. Küchlin, SOME REDUCTION STRATEGIES FOR ALGEBRAIC TERM REWRITING, Sigsam. Bull. (USA), Vol. 16, no. 4, 13-23, 1982

[26] K. Sakai, KNUTH-BENDIX ALGORITHM FOR THUE SYSTEM BASED ON KACHINUKI ORDERING, ICOT Technical Memorandum: TM-0087, 1987

MC-OBJ: UN INTERPRETE IN "C" PER "OBJ"

C. Cavenaghi, M. De Zanet, G. Mauri

Dipartimento Scienze dell'Informazione

Università di Milano

RIASSUNTO

In questo articolo si illustrano le principali caratteristiche di un interprete, sviluppato presso il Dipartimento di Scienze dell'Informazione dell'Università di Milano, per il linguaggio Obj. L'interprete è scritto in C e gira su Vax sotto il sistema operativo UNIX™. In particolare, si descrivono le strutture di dati usate per la rappresentazione interna delle espressioni e gli algoritmi di parsing e di valutazione (basati sulla riscrittura).

ABSTRACT

This paper illustrates the main features of an interpreter, developed at the Department of Information Sciences of the University of Milano, for the language Obj. The interpreter is written in C and runs on Vax under UNIX™ operating system. In particular, the data structures used for the internal representation of expressions and the algorithms of parsing and evaluation (based on rewriting) of expressions are described.

1. INTRODUZIONE

In questo articolo viene illustrato un nuovo interprete per Obj realizzato in C presso il Dipartimento di Scienze dell'Informazione di Milano, chiamato MC-Obj.

Obj è un linguaggio di specifica eseguibile sviluppato da J.A. Goguen e basato sulla logica equazionale. Tra le caratteristiche peculiari del linguaggio ricordiamo la interattività, l'approccio (di tipo bottom-up) gerarchico e modulare al processo di specificazione, la presenza di una semantica denotazionale di tipo algebrico, la possibilità di definire tipi di dati astratti anche parametrici, di trattare condizioni eccezionali e di testare direttamente le specifiche costruite, interpretando le equazioni come regole di riscrittura. Per una più completa descrizione del linguaggio rimandiamo a [3], [8] e [6].

La scelta, per l'implementazione, di un linguaggio non funzionale ma imperativo, quale è C, teoricamente poco adatto ad interpretare linguaggi non procedurali come Obj, è dovuta essenzialmente a considerazioni di efficienza. L'uso di C consente infatti di rendere più veloci la fase di parsing dei moduli e soprattutto la fase di valutazione. Inoltre, l'interprete è stato implementato sotto UNIX™, di cui il C è il linguaggio di sistema. Ciò ha consentito di sfruttare a pieno gli strumenti messi a disposizione dal sistema operativo e di fornire quindi all'utente un buon ambiente di lavoro con l'implementazione completa ed efficiente dei comandi di utilità per il trattamento e l'editing dei files di specifiche senza la necessità di disegnare editors ed algoritmi di deparsing ad hoc.

L'utilizzo di C ha comportato peraltro l'uso di tecniche e soluzioni implementative completamente diverse da quelle adottate nei precedenti interpreti in Lisp. Nei successivi paragrafi vedremo in particolare i due algoritmi di parsing delle espressioni e di riscrittura assieme ad una descrizione delle strutture dati che entrambi condividono.

La versione di Obj cui l'interprete si riferisce è ObjT, prima versione del linguaggio implementata da J. Tardo presso l'UCLA. Tuttavia sono state introdotte diverse modifiche rispetto all'interprete realizzato da Tardo. In particolare sono stati ridisegnati i moduli built-in (con un diverso trattamento della sorta "id" degli identificatori), sono stati introdotti nuovi comandi di utilità, è stato migliorato il livello di interattività ed è stato radicalmente

cambiato il modo di utilizzare le sottosorte. Anche la sintassi del linguaggio è stata leggermente modificata introducendo nuove keyword che a nostro parere rendono le specifiche più leggibili. La forma dei due costrutti **obj** ed **image,** che specificano rispettivamente oggetti e istanze di oggetti parametrici, è la seguente:

obj <nome-oggetto> [**extending** <lista-di-nomi>] is
  sorta <lista-di-nomi> [**using** <lista-di-nomi>]
  ok-ops
    <pattern>: <dominio$_1$> ··· <dominio$_n$> → <codominio> [(<lista-di-attributi>)]
      .....
  err-ops
      .....
  fix-ops
      .....
  vars
      <lista-di-nomi>: <sorta>
      .....
  ok-eqns
      [**as**<sorta>:] (<membro-sinistro>= <membro-destro> [**if** <condizione>])
      .....
  err-eqns
      .....
  eqns
      .....
endobj

image <nome-sorgente> **as** <nome-target>
    [**extending** <lista-di-nomi>] is
  sorts
      <nome-sorta-sorgente> **to** <nome-sorta-target>
      .....
  ops
      <pattern-sorgente>: <dominio> → <codominio> **to** <pattern-target>
      .....
  vars
      .....
  ok-eqns
      .....
  err-eqns
      .....
  eqns
      .....
endimage

L'interprete per ora non gestisce operatori di tipo associativo e commutativo ed inoltre è sprovvisto del comando **check** per il test di Knuth-Bendix sulla confluenza locale del sistema di riscrittura indotto dalle equazioni ([5] e [4]).

Il codice gira attualemnte su VAX 11/750 presso il Centro di Calcolo dell'Università di Milano; è in corso la realizzazione di una seconda release con trasporto di quest'ultima anche su Micro-Vax e Mc-Intosh.

2. LA STRUTTURA DEI DATI

L'intreprete è costituito essenzialmente da un algoritmo di parsing, che analizza le specifiche introdotte dall'utente, generando man mano l'ambiente al cui interno verranno valutate le espressioni (programmi), e da un valutatore di espressioni basato su un algoritmo di riscrittura.

Questi due algoritmi utilizzano una struttura dati comune che viene costruita dal parser mano a mano che l'utente immette la sua specifica ed è poi sfruttata dal parser stesso per effettuare una serie di controlli semantici sui moduli introdotti e dall'algoritmo di riscrittura per calcolare la forma delle espressioni da testare.

La struttura deve essere tale, da un lato, da non appesantire eccessivamente il parsing dei moduli, facilitando i relativi controlli di correttezza e minimizzando la ridondanza di informazioni, e, dall'altro, da favorire l'algoritmo di valutazione rendendo disponibile nel minor tempo possibile tutti i dati ad esso necessari. Per raggiungere questi scopi si è cercato anzitutto di creare delle strutture dati omogenee, adatte a trattare sia oggetti che immagini. Ciò rende più facile la stesura degli algoritmi di attraversamento del data-base interno e soprattutto rende più semplice l'implementazione delle nozioni di ambiente globale e locale di valutazione. Si è cercato inoltre di compattare ove possibile le informazioni minimizzando la quantità di memoria necessaria. È stata tuttavia introdotta ridondanza in tutte le strutture dati direttamente utilizzate dal valutatore al fine di renderlo il più rapido possibile.

2.1 Rappresentazione interna di un oggetto

Le informazioni definite all'interno di un oggetto sono: nome, sottooggetti, sorte, sottosorte, operatori, variabili ed equazioni. Tutte queste informazioni devono essere codificate in una forma interna in quanto indispensabili per poter effettuare i controlli necessari sui moduli introdotti, creare l'ambiente locale e soprattutto per poter attivare poi l'algoritmo di valutazione. La struttura scelta è piuttosto complessa ma regolare ed omogenea: essa consiste essenzialmente di liste di record con puntatori ad altre liste di record (si veda la Figura 1). Una costante fondamentale di questa struttura è quella di memorizzare a partire da un certo oggetto solo le informazioni sintatticamente legate ad esso. Ciò comporta la presenza in punti diversi del data base interno di operatori ed equazioni relativi alla medesima sorta, ma definiti in oggetti diversi, ma permette una gestione molto più semplice della nozione di ambiente locale.

La lista principale (Figura 1) accoda i moduli che l'utente introduce durante la sessione di lavoro, rappresentati da record, detti **objrec.** L'oggetto corrente viene sempre posto in cima alla lista in modo che possa essere facilmente deallocato in caso di errore o richiesta di editing da parte dell'utente durante la fase di specificazione.

Ogni **objrec** contiene il nome dell'oggetto cui si riferisce, i nomi degli antenati eventuali che esso arricchisce, le variabili con i nomi delle sorte relative e due puntatori: uno alla lista delle sorte e l'altro al record successivo. Si è deciso di memorizzare a questo livello i nomi delle variabili perchè esse sono locali all'oggetto, non sono cioè né esportabili ad altri moduli né importabili da altri mo-

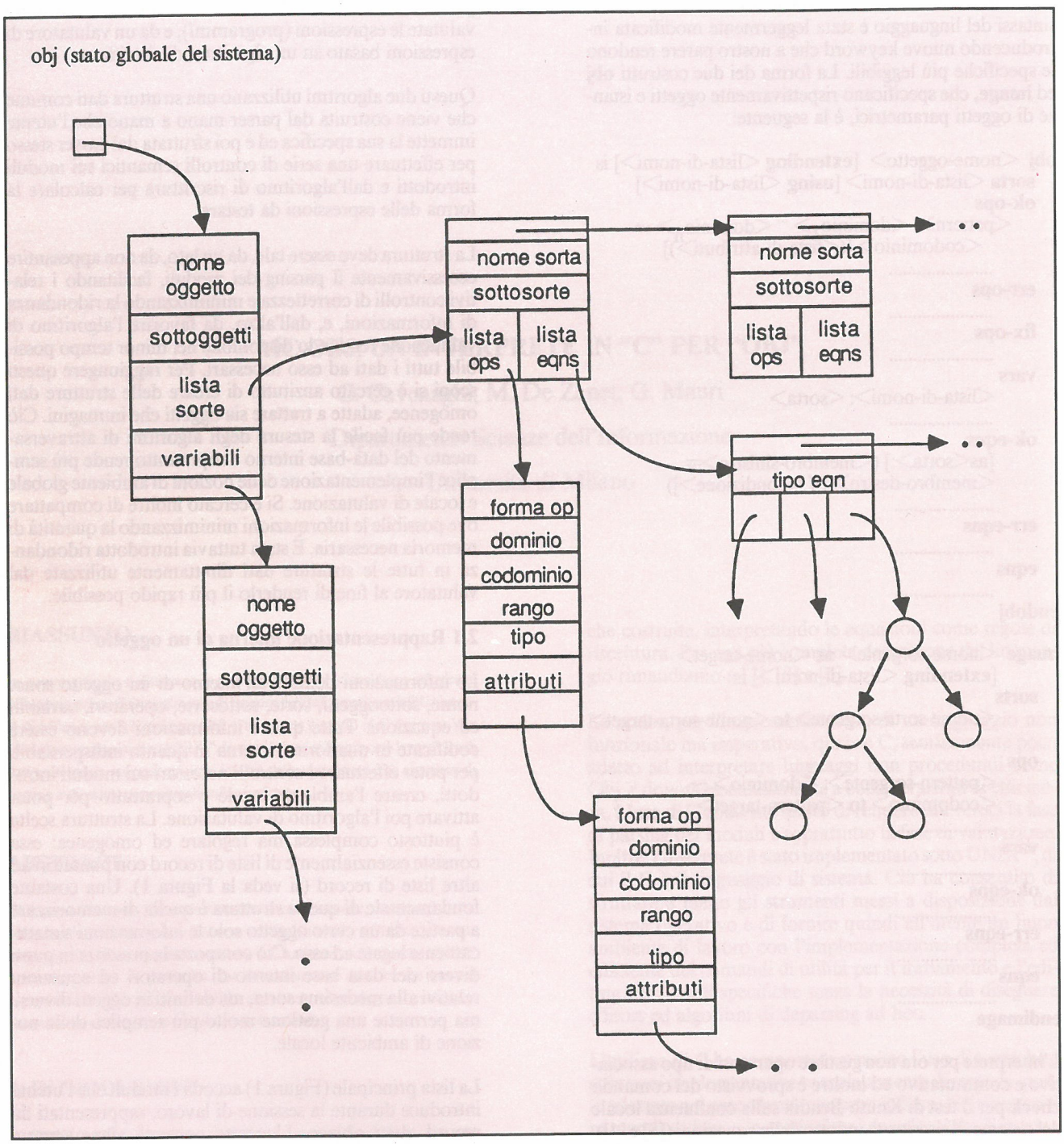


Fig. 1

duli. Il campo di **objrec** che contiene queste variabili viene definito come un array di records **varrec.** Ogni **varrec** contiene a sua volta due campi: il primo memorizza il nome della sorta cui le variabili appartengono, il secondo i loro nomi.

La lista delle sorte definite in un oggetto, cui punta il relativo **objrec,** è costituita da records **sortrec,** ognuno dei quali contiene il nome di una sorta e tre puntatori: il primo alla lista degli operatori, il secondo alla lista delle equazioni e l'ultimo alla successiva sorta della lista. Gli operatori memorizzati a partire da **sortsrec** sono quelli che hanno la sorta come codominio, cioè che nella loro applicazione portano a termini appartenenti ad essa. Le equazioni inserite sono invece quelle che riscrivono termini appartenenti alla sorta. All'inteno di ogni **sortsrec** vi è infine un campo, definito come un array di nomi di sorte, in cui sono memorizzati quegli operatori particolari

che sono le coercizioni il trattamento di questi operatori, rappresentando all'interno dell'interprete un problema particolare, verrà illustrato in un paragrafo successivo.

Veniamo ora alla struttura concernente gli operatori, costituita da records **opsrec** contenenti anzitutto la definizione del pattern sintattico dell'operatore, con trattini ad indicare la posizione dei vari argomenti. Vi sono poi quattro campi particolari, ciascuno di un bit, utilizzati per memorizzare le eventuali proprietà (hidden, comm, assoc, idmpt) dell'operatore. Viene qui sfruttata la capacità della versione del C utilizzata di operare su bit-fields definiti dall'utente, che consente di compattare lo spazio richiesto per la memorizzazione di certe informazioni. Altri due bit-fields, rispettivamente di tre e due bit, sono utilizzati per ricordare il rango dell'operatore ed il suo tipo (ok, err o fix). Infine, abbiamo la eventuale precedenza dell'operatore, i nomi delle sorte che compongono il suo dominio e codominio e l'indirizzo al successivo operatore in lista.

La lista delle equazioni è costituita da records **eqnsrec** più semplici dei precedenti; ognuno di essi contiene infatti solo quattro indirizzi ed un bit-field. Quest'ultimo è un campo di due bit il cui contenuto identifica il tipo dell'equazione (ok-eqn, err-eqn o eqn). I primi tre indirizzi sono puntatori a nodi, la cui struttura verrà chiarita più avanti, ed accedono rispettivamente al membro sinistro, al membro destro ed alla eventuale condizione dell'equazione. Il quarto puntatore contiene l'indirizzo della successiva equazione in lista.

2.2 Rappresentazione interna di un'immagine

In Obj ogni oggetto definito è parametrico, nel senso che le sorte e gli operatori forniti al momento delle specificazione possono essere sempre visti come parametri e quindi istanziati in seguito mediante il costrutto **image.** Chiameremo *oggetto sorgente* l'oggetto parametrico da istanziare e *oggetto bersaglio* l'oggetto istanziato.

Per definire una struttura dati per la rappresentazione interna di un'immagine abbiamo tenuto conto dei seguenti fatti:

— non tutte le sorte dell'oggetto sorgente sono necessariamente istanziate in quello bersaglio. Le sorte non istanziate sono riportate tali e quali. Non è però possibile creare nuove sorte se non come istanziazioni di sorte sorgenti.

— la stessa osservazione vale anche per gli operatori. Inoltre più operatori possono essere istanziati in un unico operatore bersaglio.

— le variabili e le equazioni di un oggetto sorgente sono automaticamente importate nell'immagine. Tuttavia l'utente può dichiarare nuove variabili in quanto può aggiungere nuove equazioni all'oggetto istanziato.

— tutte le sorte, gli operatori e le equazioni presenti negli antenati non comuni agli oggetti sorgente e bersaglio vanno trasportati nell'ambiente locale di quest'ultimo.

La soluzione implementativa scelta in MC-Obj consiste nel creare, durante l'analisi di un'immagine, una struttura dati del tutto identica a quella ottenuta con il parsing di un oggetto ad essa equivalente. Ciò implica la copiatura fisica di tutti i record dell'oggetto sorgente e delle sorte, degli operatori, delle variabili e delle equazioni degli antenati non comuni. Queste importazioni devono essere effettuate prima che l'analisi delle immagini venga iniziata, in quanto solo così l'attualizzazione può avvenire sostituendo i nomi formali dell'oggetto sorgente con quelli attuali dell'oggetto bersaglio. Si può intuire come tutto ciò sia molto dispendioso sia in termini di tempo che in termini di spazio. In termini di tempo in quanto la copiatura viene effettuata prima che venga stabilita la correttezza dell'immagine da analizzare, con il rischio di dover poi deallocare tutte le strutture allocate in caso di errore irreparabile. In termini di spazio in quanto si perde completamente la concisione del costrutto **image** e vengono importate anche sorte ed operatori non attualizzati, con creazione di ridondanza nel data base interno.

Questa implementazione è però estremamente comoda per l'algoritmo di valutazione che si trova ad avere subito a disposizione, già aggiornati, tutti gli operatori e le equazioni relativi all'oggetto bersaglio. Inoltre in tal modo la rete semantica che si viene a creare risulta omogenea, e non occorre introdurre strutture dati specifiche per le immagini, con conseguente complicazione degli algoritmi di attraversamento del data base interno.

La copiatura della struttura dati dell'oggetto sorgente è realizzata da una funzione **imcopy** appositamente disegnata; la versione C usata supporta l'assegnamento tra record e quindi facilita tale copiatura. La struttura dei record sopra descritta resta quindi inalterata; tuttavia d'ora in poi useremo il nome **modrec** invece di **objrec** per individuare i records contenenti la forma interna dell'intestazione sia di oggetti che di immagini.

2.3 Sorte e sottosorte

Per quanto riguarda la modalità di memorizzazione delle sorte e delle sottosorte dichiarate in ogni modulo, le alternative possibili sono essenzialmente due.

La prima, adottata da tutte le implementazioni Lisp di Obj, consiste nell'associare al modulo corrente solo le informazioni relative alle sorte che esso vede come nuove, mettendo invece i dati sulle sottosorte, cioè i nuovi operatori e le equazioni ad essi relative, nei records dei moduli antenati che le hanno dichiarate per la prima volta. Essa obbliga a ricordare in ogni record relativo ad operatori ed equazioni anche il nome del modulo in cui sono state dichiarate. Infatti la presenza di un operatore o di un'equazione in un modulo non implica più che sia stato definito proprio in esso. Lo svantaggio principale è costituito dai problemi di deallocazione delle strutture nel caso di errore da parte dell'utente o nel caso in cui questi voglia entrare in fase di editing e quindi tutto il modulo corrente debba essere scaricato. Con questa prima tecnica la deallocazione risulta estremamente laboriosa ed obbliga ad una lunga ricerca lungo tutto il data base.

La seconda soluzione, adottata in MC-Obj, mette invece nella lista delle sorte del modulo corrente anche i records relativi alle sottosorte, insieme agli operatori ed alle equazioni relative. La conseguenza principale è quella di memorizzare informazioni sulla medesima sorta in più parti del data base interno. Ciò potrebbe complicare notevolmente il parsing delle espressioni ed allungare i tempi di ricerca di operatori ed equazioni. Tuttavia, come vedremo meglio nel prossimo paragrafo, l'approccio scelto in MC-Obj per analizzare le equazioni è guidato non tanto dai nomi delle sorte di codominio quanto dai pattern degli operatori presenti nell'espressione da analizzare. Ne segue che questa dispersione di dati relativi alle varie sorte non crea alcun problema.

Spendiamo infine alcune parole sul concetto di sottosorta presente in MC-Obj. Anche qui le cose cambiano radicalmente rispetto al caso degli interpreti in Lisp. In questi ultimi infatti la dichiarazione delle sottosorte è considerata opzionale e le sorte dei sottooggetti sono importate automaticamente nel nuovo modulo. Ciò fornisce maggiore flessibilità all'utente ma comporta, per lo meno nel caso di rappresentazioni intermedie come quella usata in MC-Obj, un notevole spreco di tempo e di memoria. È stata pertanto fatta una scelta diversa, rendendo obbligatoria la dichiarazione delle sottosorte usate esplicitamente (per sorta usata esplicitamente intendiamo una sorta arricchita dal modulo corrente, cioè per la quale vengano definiti nuovi operatori o nuove equazioni che ritornano termini appartenenti ad essa). L'interprete provvede invece ad importare automaticamente le sottosorte usate in modo implicito o come dominio di un operatore o come sottoespressione di una equazione. In questo modo, la dichiarazione delle sottosorte equivale ad una vera e propria importazione, con relativi controlli semantici, dei tipi di dati presenti nei moduli antenati. Tutto ciò ha anche dei notevoli vantaggi pratici in quanto consente di utilizzare la memoria strettamente necessaria e rende più semplici certi controlli da parte dell'interprete.

2.4 Il trattamento delle coercizioni

Anche le coercizioni sono implementate in MC-Obj in maniera particolare. Ciò che infatti l'interprete memorizza nel data base interno non è la relazione di coercizione tra due sorte ma la controrelazione che dichiara una sorta essere sottoinsieme proprio di un'altra. Questa scelta è necessaria per rendere possibile il parsing delle espressioni. Consideriamo ad esempio una specifica che introduca gli operatori:

—: elem → stack
push: elem stack → stack

A questo punto per poter analizzare correttamente una espressione del tipo:

push ((.....), (.....))

rispettando la coercizione definita, bisogna essere in grado di scartare nel primo argomento tutte le interpretazioni di sorta "stack" ed accettare invece nel secondo argomento tutte le interpretazioni di sorta "elem". Se dunque la coercizione fosse trattata memorizzando il nome "stack" nel **sortsrec** di "elem", nel primo caso non sapremmo come utilizzare questa informazione e nel secondo caso essa obbligherebbe l'interprete a scandire l'intero data base alla ricerca di questo **sortsrec.** Risulta quindi molto più comodo ed utile considerare la controrelazione ed inserire il nome "elem" nel **sortsrec** di "stack". Così facendo siamo in grado di accettare tutte le interpretazioni corrette dei vari argomenti di una qualsiasi sottoespressione. In definitiva nel campo **coercion** dei vari **sortsrec** vengono memorizzati i nomi di tutti quei tipi che possono essere considerati sottoinsiemi della sorta corrente.

Questo modo di procedere è anche in sintonia con l'architettura del sistema in quanto la coercizione precedente, anche dal punto di vista sintattico, vede "stack" come codominio dell'applicazione e quindi come sorta arricchita. Ciò determina la presenza del **sortsrec** relativo nella lista delle sorte e sottosorte dell'oggetto corrente e quindi la possibilità di memorizzare tutte le informazioni a partire dal **modrec** corrente.

L'utilizzo delle coercizioni crea però anche altri problemi legati alle catene che possono crearsi tra più sorte a causa della transitività della relazione. L'interprete, oltre a riconoscere l'esistenza di queste catene, deve anche verificare la assenza di cicli che determinerebbero sequenze di riscrittura infinite. Questi problemi possono essere risolti in maniera efficiente facendo in modo che tutte le volte che una sorta viene importata da un altro modulo vengano copiati anche i sottoinsiemi fino a quel momento definiti per essa. Quando infatti abbiamo in input la dichiarazione:

x → y

l'interprete deve testare l'esistenza di "x" nell'ambiente locale corrente. Se ora facciamo in modo che questa ricerca ritorni il puntatore all'ultimo **sortsrec** allocato per "x", avremo accesso a tutti i nomi dei sottoinsiemi definiti fino a quel momento per questa sorta. A questo punto le catene che si vengono eventualmente a creare vengono risolte semplicemente importando questi nomi nel **sortsrec** di "y". In tal modo anche il test sulla presenza di cicli diventa estremamente veloce: basta infatti verificare prima di questa importazione che "y" non sia già presente tra i vari sottoinsiemi di "x".

2.5 Rappresentazione delle equazioni

La rappresentazione interna delle espressioni che compongono le equazioni in MC-Obj è ad albero (Fig. 2). Ogni nodo dell'albero, a partire dalla radice che corrisponde all'operatore più esterno, rappresenta un operatore o una variabile dell'espressione. Questa informazione viene memorizzata nel record del nodo come un puntatore all'**opsrec** o al **varrec** relativo. Il record del nodo deve inoltre tener conto del fatto che per l'algoritmo di valutazione é indispensabile poter attraversare gli alberi in entrambe le direzioni: dalla radice alle foglie e viceversa. Ciò significa dover provvedere un puntatore al nodo padre.

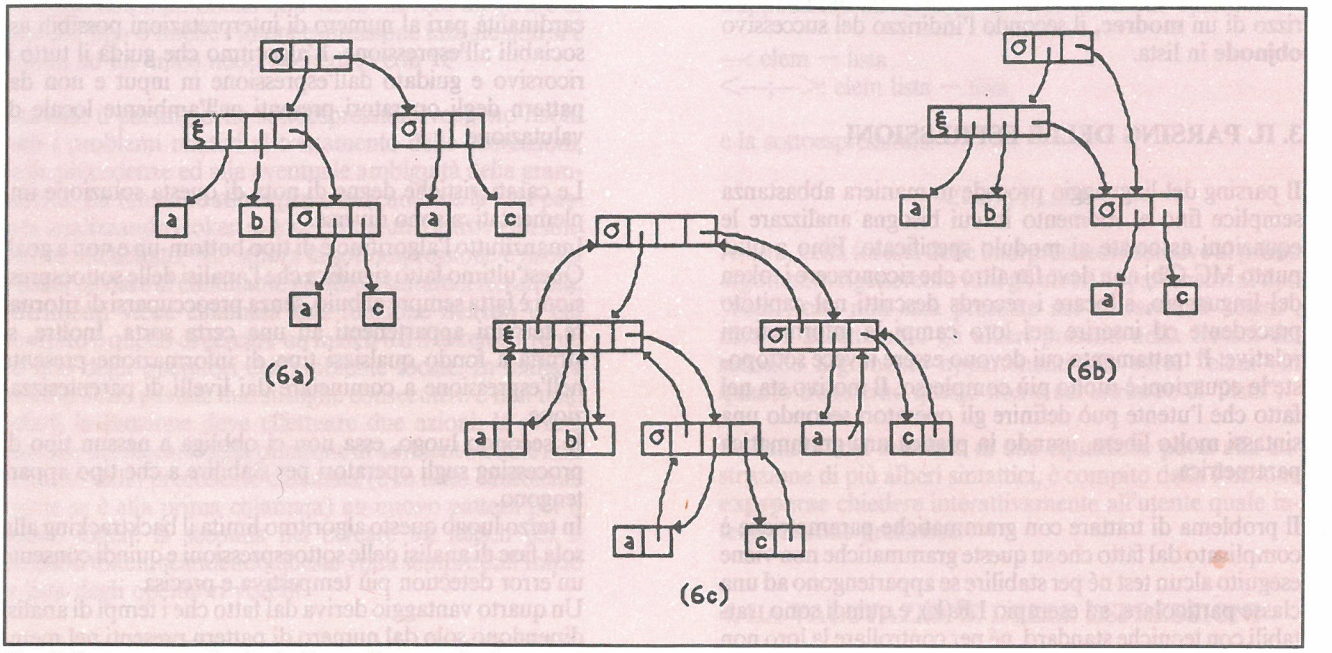


Fig. 2 Rappresentazioni ad albero di espressioni

Sempre nei nodi sono inoltre definiti quattro bit-fields. I primi due, di un bit ciascuno, tengono conto dello stato di riduzione della sottoespressione avente il nodo per radice e del tipo dell'operatore, cioè se esso è built-in (e quindi trattato mediante funzioni predefinite del C) o definito dall'utente. Gli altri sono di due bit ciascuno e ricordano lo stato d'errore del nodo e se l'operatore cui esso si riferisce era stato dichiarato di tipo ok, fix, err oppure se si tratta di una variabile. La presenza di quest'ultima informazione in tutti i nodi crea della ridondanza in quanto sarebbe stato possibile distinguere i vari tipi di operatori semplicemente inserendoli in tre liste diverse già durante la fase di parsing. Tuttavia questi bit consentono al valutatore di conoscere immediatamente lo stato di errore dell'operatore con cui ha a che fare. Ciò è importante in quanto è in base a questa informazione che l'algoritmo di riscrittura decide la strategia con cui deve applicare le varie equazioni. Con questa implementazione si evitano quindi lunghi e costosi attraversamenti di liste prima di ogni riscrittura.

Sempre per facilitare il valutatore, nel record di ogni nodo è presente un intero che ricorda la sua profondità all'interno dell'albero. Un ultimo problema è legato al possibile numero di figli di ogni nodo. Chiaramente esso dipende dall'arità dell'operatore relativo, che può essere anche elevato. Nel caso ad esempio della specifica di uno "scheduler airport" fatta da [8], compaiono operatori di arità sei e cinque. A livello della struttura dei record dei nodi non è conveniente quindi, per motivi di ottimizzazione dello spazio, creare un unico modello di nodo. È stata anche scartata l'idea di utilizzare dei record di tipo variant in quanto le "union" del C non fanno altro che allocare sempre lo spazio per il record di dimensione massima. Per minimizzare il numero dei puntatori ai figli, si è quindi deciso di creare più template di nodi, ciascuno relativo ad operatori di una ben precisa arità. L'interprete è in grado di accedere a questa informazione prima di allocare il nodo e quindi può scegliere, tra i vari record a disposizione, quello più adatto per l'operatore corrente.

2.6 Ambiente globale ed ambiente locale

La lista dei **modrec** che si viene a creare durante l'introduzione delle specifiche da parte dell'utente crea automaticamente l'ambiente globale di lavoro della sessione corrente. Lo header di questa lista è dato dall'ultimo modulo inserito e vi si può accedere mediante il puntatore di ambiente globale Obj.

Per quanto riguarda invece l'ambiente locale, raggruppa tutti gli oggetti ed immagini rispetto ai quali può essere fatta una valutazione. Esso indica cioè l'insieme di operatori ed equazioni che possono essere utilizzati dall'algoritmo di riscrittura durante il calcolo delle forma ridotta della espressione da testare. Per default questo insieme viene calcolato relativamente all'ultimo modulo inserito ed ai sottooggetti raggiungibili da esso. L'utente può comunque cambiare in qualsiasi momento l'ambiente di valutazione spostandosi in qualunque punto dell'ambiente globale mediante il comando **visit.**

Internamente, l'ambiente locale è rappresentato da una lista della **objpath.** Essa contiene i riferimenti a tutti i moduli visibili e viene manipolata da una funzione **local-env** che provvede al suo continuo aggiornamento. La presenza dei puntatori ai sottooggetti già a livello dei **modrec** rende più efficiente **local-env** che durante l'attraversamento dell'ambiente globale, a partire dal modulo corrente, può accedere in brevissimo tempo a questa informazione e costruire quindi in maniera più efficiente l'ambiente locale. I nodi della lista **objpath,** detti **objnode,**

sono provvisti di due soli campi: il primo contiene l'indirizzo di un **modrec**, il secondo l'indirizzo del successivo **objnode** in lista.

3. IL PARSING DELLE ESPRESSIONI

Il parsing del linguaggio procede in maniera abbastanza semplice fino al momento in cui bisogna analizzare le equazioni associate al modulo specificato. Fino a quel punto MC-Obj non deve far altro che riconoscere i token del linguaggio, allocare i records descritti nel capitolo precedente ed inserire nei loro campi le informazioni relative. Il trattamento cui devono essere invece sottoposte le equazioni è molto più complesso. Il motivo sta nel fatto che l'utente può definire gli operatori secondo una sintassi molto libera, usando in pratica una grammatica parametrica.

Il problema di trattare con grammatiche parametriche è complicato dal fatto che su queste grammatiche non viene eseguito alcun test né per stabilire se appartengono ad una classe particolare, ad esempio LR(k), e quindi sono trattabili con tecniche standard, né per controllare la loro non ambiguità. Ne segue che non solo l'algoritmo di parsing deve saper trattare grammatiche sempre diverse, ma deve saper anche interpretare espressioni ambigue, riconducibili cioè a più alberi sintattici.

La possibilità di creare e sfruttare grammatiche ambigue conferisce notevole potenza e flessibilità al sistema Obj soprattutto se l'algoritmo di parsing relativo è in grado di gestire interattivamente tutte le interpretazioni possibili facendo scegliere all'utente di volta in volta quella preferita.

Un'altra fase importante nella analisi delle equazioni è la traduzione dalla loro forma esterna (sequenza di caratteri) alla rappresentazione interna, come coppie di alberi n-ari aventi per radici gli operatori principali dei rispettivi membri.

I membri di ogni equazione vengono analizzati da MC-Obj dall'interno verso l'esterno, isolando dapprima le sottoespressioni, che vengono poi a loro volta analizzate con una tecnica particolare, cui d'ora in poi ci riferiremo col nome di "a-l" (accetta-lascia), che consente di costruire la foresta di tutte le interpretazioni (alberi sintattici) ad esse relative.

La tecnica "a-l" verrà illustrata più avanti; l'idea di base è qualla di intraprendere per ogni pattern trovato due strade: la prima accetta l'operatore e costruisce il sottoalbero relativo ad esso ed ai suoi argomenti, la seconda lascia l'operatore e va alla ricerca di un nuovo pattern. Ciò viene fatto per tutti i pattern possibili, effettuando un backtracking nei casi in cui si sia riusciti a costruire un albero completo per la sottoespressione oppure si sia incorsi in un errore. Man mano che si risale nei livelli di parentesizzazione dell'espressione si effettua il prodotto cartesiano di tutte le foreste costruite fino a quel momento con gli argomenti dell'operatore corrente, scartando le interpretazioni che non ritornano termini della sorta corretta. Al termine dell'elaborazione si ha così una foresta di alberi di cardinalità pari al numero di interpretazioni possibili associabili all'espressione. L'algoritmo che guida il tutto è ricorsivo e guidato dall'espressione in input e non dai pattern degli operatori presenti nell'ambiente locale di valutazione.

Le caratteristiche degne di nota di questa soluzione implementativa sono diverse.
Innanzitutto l'algoritmo è di tipo bottom-up e non a goal. Quest'ultimo fatto significa che l'analisi delle sottoespressioni è fatta sempre al buio, senza preoccuparsi di ritornare termini appartenenti ad una certa sorta. Inoltre, si sfrutta a fondo qualsiasi tipo di informazione presente nell'espressione a cominciare dai livelli di parentesizzazione.
In secondo luogo, essa non ci obbliga a nessun tipo di processing sugli operatori per stabilire a che tipo appartengono.
In terzo luogo questo algoritmo limita il backtracking alla sola fase di analisi delle sottoespressioni e quindi consente un'error detection più tempestiva e precisa.
Un quarto vantaggio deriva dal fatto che i tempi di analisi dipendono solo dal numero di pattern presenti nel membro dell'equazione. Questo algoritmo inoltre non ricalcola mai due volte la stessa sottoespressione, come può capitare invece negli algoritmi top-down.

L'algoritmo è diviso in due fasi distinte. Nella prima viene effettuata una analisi lessicale del membro dell'equazione. Durante questa analisi viene controllato il bilanciamento delle parentesi e si trasforma l'espressione dalla sua forma esterna in una rappresentazione interna come lista doppiamente linkata di records. Ognuno di questi record contiene un token ed un puntatore, inizialmente nullo, ad una foresta di alberi. Il token può essere di volta in volta una virgola, una parentesi, una variabile oppure parte di un operatore. Un token speciale, ad esempio la chiocciola {@}, indica invece che il puntatore non è nullo ma va ad una foresta di interpretazioni, cioè che il record sta al posto una sottoespressione già risolta. Le variabili e gli operatori costanti vengono trattati già a questo livello creando per essi il record "@" con il relativo unico albero sintattico. Ciascun nodo degli alberi contiene le informazioni descritte nel capitolo precedente.

Nella seconda fase avviene l'analisi semantica vera e propria. Una prima funzione **exprparse** tratta l'intero membro dell'equazione analizzando i livelli di parentesizzazione in esso presenti. Quando **exprparse** trova una parentesi aperta memorizza il puntatore all'elemento della lista corrente in uno stack. Quando trova una parentesi chiusa isola la sottoespressione così individuata, crea la nuova lista relativa ad essa e lancia una seconda funzione **subexprparse** che effettua il parsing di quest'ultima. A questo punto **subexprparse** è in grado di separare i vari argomenti eventualmente presenti nella sottoespressione semplicemente individuando le virgole in essa contenute. Ogni argomento quindi analizzato con la tecnica "a-l". La funzione **subexprparse** ritorna un nodo etichettato "@" con la foresta delle interpretazioni. Questo nodo va a sostituirsi nella lista di partenza alla sottoespressione. Il

processo continua finché non viene rilevato un errore di qualche tipo o finché l'intera espressione iniziale non si è ridotta ad un unico nodo etichettato con "@".

Durante il parsing delle sottoespressioni vengono risolti tutti i problemi relativi al trattamento delle coercizioni, delle precedenze ed alla eventuale ambiguità della grammatica. La funzione **subexprparse** scandisce la lista passata analizzando i token in essa contenuti. Se trova un solo record etichettato "@", allora la sottoespressione è risolta e basta passare al chiamante gli alberi sintattici ricostruiti. Altrimenti viene chiamata una funzione **nextop** il cui compito è quello di cercare un match tra il token corrente ed uno degli operatori dell'ambiente locale. In realtà a **nextop** sono passate due stringhe consecutive e non una. Infatti la funzione deve effettuare due azioni: la prima consiste nel cercare nella porzione di ambiente locale non scandito dalla precedente chiamata (o in tutto l'ambiente locale se è alla prima chiamata) un nuovo pattern per il primo token, la seconda nel cercare un match per il secondo token, scandendo questa volta sempre dall'inizio la lista degli operatori visibili.

La necessità della prima azione deriva dal fatto che l'utente può aver definito più operatori di tipo misto col primo token in comune; caso tipico a questo riguardo sono glioperatori condizionali if_then e if_then_else. Per ogni operatore ritornato si intraprendono in **subexprparse** due strade. Nella prima si accetta l'operatore e si crea un nuovo nodo "@" che punta a tanti alberi sintattici quanti sono gli elementi (compatibili con le arità e le coercizioni) dell'insieme formato dal prodotto cartesiano degli argomenti dell'operatore. Si costruisce quindi una nuova lista e si richiama ricorsivamente **subexprparse** su quest'ultima. La seconda strada non accetta l'operatore ma richiama **nextop** per una nuova ricerca.

Si noti che nel momento in cui si accetta un operatore non si ipotizza che esso sia principale come negli approcci top-down; inoltre, la funzione **subexprparse** provare tutte le associatività possibili all'interno della sottoespressione, e proprio questa sua caratteristica la rende particolarmente adatta ad interpretare grammatiche ambigue.

La scelta di un operatore è anche legata alla sua precedenza: **subexprparse** viene sempre chiamata da **exprparse** con precedenza massima in quanto, essendo l'analisi delle sottoespressioni di tipo bottom-up, gli alberi sono costruiti a partire dalle foglie e quindi gli operatori via via riconosciuti devono avere precedenza minore o uguale a quella dell'ultimo pattern trovato.

La presenza delle coercizioni non complica di molto l'algoritmo finora descritto. Basta infatti leggere, mentre si considerano tutte le combinazioni di interpretazioni relative agli argomenti di un operatore, l'array presente in tutti i **sortsrec** dei nomi dei tipi che possono essere visti come sottosistemi della sorta corrente per conoscere quali interpretazioni possono essere valide per un certo albero sintattico.

Supponiamo ad esempio di avere i soli due operatori:

—: elem → lista
<—;—>: elem lista → lista

e la sottoespressione:

<(.....) , (.....)>

Allora, nella foresta delle interpretazioni relative al primo argomento rigetteremo tutti gli alberi appartenenti al tipo "lista", che non sarà presente nel **sotrsrec** di "elem", mentre accetteremo gli alberi presenti nella foresta del secondo argomento appartenenti alla sorta "elem" in quanto troveremo questo nome nel **sortsrec** di "lista".

Se l'analisi del membro di una equazione porta alla costruzione di più alberi sintattici, è compito della funzione **exprparse** chiedere interattivamente all'utente quale interpretazione preferisce.

4. LA VALUTAZIONE DELLE ESPRESSIONI

In questo paragrafo descriviamo l'implementazione del valutatore di espressioni in MC-Obj. Poiché le espressioni vengono valutate attraverso una tecnica di riscrittura, dovremo:

1) sviluppare un algoritmo efficiente per trovare i sottotermini riscrivibili (redex);

2) determinare una strategia di riscrittura per scegliere il redex da rimpiazzare ad ogni costo;

3) trovare un algoritmo per eseguire il singolo rimpiazzamento.

Esaminiamo ora più in dettaglio i singoli passi mostrando via via le soluzioni scelte nell'implementazione.

Il problema di sviluppare un buon algoritmo per trovare i redex è cruciale per l'implementazione efficiente di un sistema di riscrittura. Essenzialmente, il problema di trovare i redex è un problema di "pattern matching" di alberi. La decisione di quale algoritmo di pattern matching applicare deve essere presa accuratamente, tenendo conto di eventuali proprietà speciali dei patterns e degli alberi considerati, del numero di differenti alberi attesi per il medesimo insieme di patterns, ed infine delle risorse di tempo e di spazio disponibili.

A tale riguardo in Obj, ma ciò può essere detto in generale per qualsiasi linguaggio di programmazione equazionale, si possono fare due importanti considerazioni:

i) i patterns sono le parti sinistre delle equazioni, conosciute in anticipo dal sistema, mentre l'albero soggetto è l'espressione da valutare, che è conosciuta dal sistema solo a run-time. Inoltre, il medesimo insieme di equazioni (programma) è tipicamente usato ripetutamente per valutare (tramite i comandi **run** e **rum**) differenti espressioni.

ii) ogniqualvolta un redex viene rimpiazzato, l'albero soggetto cambia "localmente", nel senso che solo una porzione dell'albero soggetto è influenzato da questo rimpiazzamento.

La prima considerazione suggerisce l'idea di dividere l'algoritmo di pattern matching (e quindi l'implementazione del sistema di riscrittura) in due parti: un pre-processor, che lavora sulle equazioni (patterns), indipendentemente dall'espressione da valutare, generando informazioni per guidare il processo di matching vero e proprio; e un programma run-time che esegue, sulla base delle informazioni generate dal pre-processor, le reali riduzioni secondo una propria strategia di riscrittura (Fig. 3). La fase di pre-processing viene attivata dall'utente entrando nella fase di valutazione delle espressioni mediante il comando **eval.**

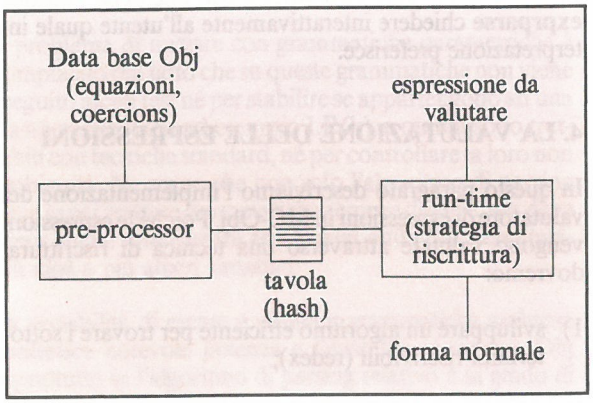


Fig. 3 Implementazione del valutatore di espressioni

La seconda considerazione ci dice che ogni riscrittura altera localmente l'albero rappresentante il termine da valutare: il nostro algoritmo di pattern matching dovrebbe essere quindi in grado di rispondere incrementalmente a tali cambiamenti locali dell'albero soggetto, evitando di riesaminare l'intero albero.

Il problema di pattern matching di alberi è stato studiato diffusamente, con particolare attenzione ai problemi legati all'implementazione dei sistemi di riscrittura, in [7], anche se non per le segnature a più sorte. Nella nostra implementazione l'enfasi è stata posta sulla minimizzazione dei tempi di matching: si è quindi scelto un approccio bottom-up caratterizzato da tempi di pre-processing relativamente alti ma da tempi di matching più veloci e da una buona risposta ai cambiamenti locali. L'idea chiave dell'approccio bottom-up per individuare i redex è di associare ad ogni nodo dell'albero soggetto un codice indicante quali sottotermini delle parti sinistre delle equazioni uguagliano il sottoalbero a quel nodo. Questi "match states" (rappresentati come stringhe di bit) sono computati nel corso del matching, a partire dalle foglie dell'albero soggetto, usando delle tavole Hash costruite durante la fase di pre-processing: esse forniscono lo stato del nodo n come funzione del simbolo della segnatura associato ad n e dello stato dei figli di n. In questo modo tutti i redex presenti nell'espressione da valutare possono essere individuati in un tempo proporzionale al numero di nodi dell'albero soggetto. Quando viene eseguita una riduzione, i "match states" devono essere computati per tutti i nuovi nodi aggiunti e per qualche nodo superiore del redex.

Il problema di generare efficientemente queste tabelle è stato studiato in [7]: tali risultati sono stati da noi ripresi ed estesi in modo tale da includere segnature a più sorte ed un corretto trattamento delle coercizioni; infine è stata sviluppata interamente l'idea di "bottom-up matching with bit-string operations" suggerita in [7].

Accenniamo ora brevemente all'algoritmo di pre-processing. Cominciamo con la seguente:

Definizione - Sia PF l'insieme di tutti i sottoalberi dei patterns che sono membro sinistro di qualche equazione dell'ambiente locale. Per ogni simbolo δ della segnatura definiamo un insieme $U\delta$ come segue:

i) se $\delta \epsilon \Sigma_{\lambda,s}$ e not($\delta \epsilon$PF), allora $U\delta=\{v_s\} \cup \{vs' | s \rightarrow * s'\}$;

ii) se $\delta \epsilon \Sigma_{\lambda,s}$ e $\delta \epsilon$PF, allora $U\delta=\{\delta, v_s\} \cup \{vs' | s \rightarrow * s'\}$;

iii) se $\delta \epsilon \Sigma_{s1s2...sq,s}$, allora $U\delta=\{t \epsilon PF | t \epsilon \delta(t_1,t_2,...,t_q)\} \cup \{v_s\} \cup \{v_s' | s \rightarrow *s'\}$.

Inoltre, dato un qualsiasi insieme di subpatterns M, definiamo la seguente funzione:

$FATHER_i(M)=\{t' \epsilon PF |$ son_i (t') è in M$\}$, cioè $FATHER_i(M)$ non è altro che l'insieme di tutti i subpaterns in PF il cui i-esimo figlio è in M. Una volta definiti gli insiemi $U\delta$ e le funzioni $FATHER_i$, possiamo dire che:

i) $Match(\delta) = U\delta$ se $\delta \epsilon \Sigma_{\Sigma,s}$;

ii) $Match(\delta(t_1,t_2,...,t_q)) = (U\delta \cap FATHER_1(Match(t_1)) \cap FATHER_2(Match(t_2)) \cap ... \cap FATHER_q(Match(t_q))) \cup \{v_s\} \cup \{vs' | s \rightarrow *s'\}$ se $\delta \epsilon \Sigma_{s1s2...sq,s}$ $(q>O)$.

Ora, una volta calcolati i $FATHER_i$, possiamo assegnare i match sets ad ogni nodo dell'albero soggetto semplicemente attraversandolo in ordine differito ed applicando le regole (i) e (ii). Il calcolo diretto dei $FATHER_i$ sembra richiedere l'esame di tutti i subpatterns. In [7] viene suggerito perciò un approccio hashing. L'idea è di calcolare a priori una hash table per tutti i match sets M e memorizzare i $FATHER_i(M)$ nell'entry della tabella corrispondente al match set M. Non è difficile dimostrare, nel caso di foreste semplici, che la funzione hash, non facendo altro che contare il numero di bit iniziali posti a zero in un match set M, è funzione hash "perfetta" minimale (si fa notare che la determinazione di una funzione hash "perfetta" è legata alla possibilità di enumerare gli elementi di PF secondo l'ordine parziale indotto da una relazione di inclusione ">" su PF). Il seguente algoritmo mostra come sia possibile precomputare efficientemente tutti i match sets in modo tale che la funzione sopra definita garantisca un hashing perfetto minimale.

Algoritmo di costruzione di tutti i possibili match sets di una foresta semplice, in modo tale che la funzione ω produca un hashing "perfetto".

input: una foresta di patterns semplice F
output: l'insieme di tutti i possibili match sets per F.

algoritmo:
<lista tutti gli elementi di PF in ordine crescente di profondità>;
for each $p \epsilon PF$, di profondità h, incrementando la profondità **do**
begin
1: $Mp:=\{p\}$;
for each $p' \epsilon PF$, di profondità minore od uguale ad h **do**
if $(p=v_s$, **and** $p'=v_s$, **and** $s \rightarrow^* s')$ **or**
$(p=\delta(p_1,p_2,...,p_q)\{q \geq O, \delta \epsilon \Sigma_{s1s2...sq,s}\}$ **and** $((p'=v_s)$
or $(p'=v_{s'}$ **and** $s \rightarrow^* s')))$
or $(p=\delta(p_1,p_2,...,p_q)$ **and** $p'=\delta\ (p'_1,p'_2,...p'_q)$
and $1 \leq i \leq q : M_{p'i} \leq M_{pi})$
then
if $ord(p)<ord(p')$ **then begin**
<i patterns p ep' sono scambiati di posto nell'enumerazione corrente degli elementi di PF>;
goto 1
end
else
$M_p:=M_p \cup \{p'\}$
end.

Una volta trovati i match sets non ci resta altro che computare per ognuno di essi gli insiemi $FATHER_i$:

Algoritmo di costruzione tavola hash contenente gli insiemi $FATHER_i$:

input: l'insieme di tutti i possibili match sets di una foresta semplice P calcolati dall'algoritmo precedente.
output: una tavola hash T per i match sets tale che per ogni match set M_p, $p=\delta(p_1,p_2,...,p_q)$, $T[\omega(Mp)]:=<FATHER_1(M_p),...,FATHER_{rank}(M_p)>$, dove ω è la funzione hash precedentemente definita.

algoritmo:
for each M_p, $p=\delta(p_1,p_2,...,p_q)$, $q \geq o$ **do**
for each $i \epsilon \{1,2,...,rank\}$ **do begin**
$FATHER_i(M_p):=\Phi$;

for each $p' \epsilon PF$ **do**
if $son_i(p') \epsilon M_p$ then $FATHER_i(M_p):=FATHER_i(M_p) \cup \{p'\}$;
$T[\omega(M)]:=<FATHER_1(M_p),...,FATHER_{rank}(M_p)>$
end.

Per quanto riguarda l'algoritmo di riduzione run-time, la strategia di riscrittura implementata è stata influenzata dalla necessità di convergere ad una forma normale ogniqualvolta il sistema di riscrittura indotto da una specifica non sia a terminazione finita.

Tenendo conto di questi due fattori l'algoritmo cerca dapprima di applicare, in successione, tutte le equazioni d'errore e di tipo fix ai sottoalberi non ancora ridotti. Dopo di ciò, i termini non manifestamente in errore sono considerati ok e ad essi sono applicate le eventuali equazioni ok. Come si è detto la riscrittura di un termine ok t è sempre vista come un tentativo: se t viene successivamente dimostrato essere un termine d'errore {applicando ricorsivamente a t l'algoritmo}, allora l'algoritmo evita di eseguire la riscrittura, cercando di applicare a t una nuova equazione di tipo ok {"backtracking"}. Una volta ridotto un termine ok, questo stesso ordine di applicazione delle equazioni è seguito per ogni nuovo albero t ottenuto dall'algoritmo.

Una volta scelto il redex da ridurre, l'esecuzione materiale della riduzione è estremamente semplice. Il parser provvede per ogni equazione ad una rappresentazione ad albero del termine destro: una copia di questa rappresentazione viene sostituita, nell'espressione da valutare, al redex scelto, rimpiazzando ogni variabile con l'appropriata sottoespressione.

Ricapitolando quanto detto, la fase di preprocessing ha inizio nell'istante in cui l'utente entra nell'ambiente di valutazione delle espressioni tramite il comando **eval.** Se il comando **eval** ha come argomento il nome di un modulo precedentemente inserito nel sistema, viene prima richiamata una funzione che provvede a ricostruire il corrispondente ambiente locale. In questa fase di pre-processing vengono generate, a partire dalle equazioni appartenenti all'ambiente locale corrente ed indipendentemente dall'espressione da valutare, le informazioni necessarie a guidare l'algoritmo di matching vero e proprio. La parte sinistra di ogni equazione dell'ambiente locale e, ricorsivamente, ogni sua sottoespressione sono poste, qualora non fossero già presenti, in una lista rappresentante la foresta PF. Tale lista è contemporaneamente ordinata, con un algoritmo di bucket sort, secondo la profondità degli alberi. Una volta enumerati in tal modo gli elementi di PF, si provvede a generare l'insieme di tutti i possibili match set per F secondo l'algoritmo visto. Successivamente un'opportuna funzione costruisce la tavola hash contenente gli insiemi Father e la lista dei match completi associati ai vari match set. Al termine di questa fase, per ogni operatore δ di sorta s appartenente all'ambiente locale vengono computati gli insiemi $U\delta$ e $\{v_s\} \cup \{v_s, | s \rightarrow^* s'\}$.

La valutazione di un'espressione (comandi **run** e **rum**) viene portata a termine in tre passi principali:

1) analisi dell'espressione da ridurre, con il raggiungimento di una rappresentazione interna.

2) riduzione dell'espressione, secondo gli algoritmi di riscrittura e di matching descritti, manipolando le sole rappresentazioni interne.

3) conversione del risultato nella sua rappresentazione sintattica esterna e sua stampa.

Se il risultato dell'analisi di un'espressione non è unico l'interprete, dopo aver stampato alcune informazioni sulle interpretazioni trovate, instaura un dialogo interattivo con l'utente per determinare quale risultato del parsing valutare: l'utente può scegliere di valutare una o tutte le interpretazioni ottenute. Ora, una alla volta, le interpretazioni dell'espressione da valutare sono passate ad una funzione **reduce** la quale è incaricata, sulla base delle informazioni generate durante la fase di pre-processing e secondo la strategia di riscrittura delineata, di trasformare la rappresentazione interna del termine di partenza nella rappresentazione interna del termine ridotto {se esiste}.

Il risultato della riduzione deve essere perciò trasformato, tramite una funzione di deparsing, nella sua rappresentazione sintattica esterna. Prima di visualizzare tale rappresentazione, l'interprete esamina lo stato d'errore dell'operatore principale: qualora abbia valore "T", cioè la forma normale è un termine d'errore, viene visualizzata la caratteristica parola chiave che indica all'utente lo stato d'errore del termine. La funzione di deparsing è chiamata ricorsivamente sugli argomenti della rappresentazione interna del termine e, utilizzando il legame presente nel sistema tra il nome interno di un operatore e la sua forma, costruisce la rappresentazione esterna del termine. Nel caso si trovi un operatore con l'attributo hidden, viene fornito un risultato nullo ed un messaggio d'errore (dal momento che questo operatore non deve essere visibile all'esterno).

Ogni operatore built-in è memorizzato nel sistema come un normale operatore definito dall'utente. Tuttavia le equazioni relative agli oggetti built-in non sono implementate equazionalmente: esse sono implicitamente specificate da opportuni frammenti di codice attivati dall'algoritmo di riscrittura. L'oggetto built-in Nat è invece implementato equazionalmente.

5. RINGRAZIAMENTI

Gli autori intendono ringraziare il prof. G. Degli Antoni per il continuo interessamento ed i preziosi consigli.

L'attività si è svolta nell'ambito del Communication and Programming Project della Università degli Studi di Milano e della HISI - Honeywell Information Systems Italia.

6. BIBLIOGRAFIA

[1] C. Cavenaghi, IMPLEMENTAZIONE DI UN VALUTATORE DI ESPRESSIONI PER IL LINGUAGGIO OBJ, Tesi di laurea, Università di Milano, 1985

[2] M. De Zanet, UN ALGORITMO DI PARSING, CREAZIONE E MANUTENZIONE DEGLI OGGETTI IN LINGUAGGIO C PER IL LINGUAGGIO OBJ, Tesi di laurea, Università di Milano, 1985

[3] J.A. Goguen, SOME DESIGN PRINCIPLES AND THEORY FOR OBJ-O, A LANGUANGE FOR EXPRESSING AND EXECUTING ALGEBRAIC SPECIFICATIONS OF PROGRAMS, Proc. International Conference on Mathematical Studies of Information Processing, Kioto, Giappone, pp. 429-475, 1978

[4] G. Huet, CONFLUENT REDUCTIONS: ABSTRACT PROPERTIES AND APPLICATIONS TO TERM REWRITING SYSTEMS, J.ACM 27, 797-821, 1980

[5] D.E. Knuth, P. Bendix, SIMPLE WORD PROBLEMS IN UNIVERSAL ALGEBRAS, Proc. Conf. on Computational problems in abstract algebra (J. Leech ed.), Pergamon Press, Oxford, 1967

[6] G. Mauri, OBJ, UN LINGUAGGIO DI PROGRAMMAZIONE BASATO SULLA LOGICA EQUAZIONALE, Note di Software, 21/22, 1983

[7] M.J. O'Donnel, C.M. Hofmann, PATTERN MATCHING IN TREES, ACM Transaction on Programming Languages and Systems, 1982

[8] J.J. Tardo, THE DESIGN, SPECIFICATION AND IMPLEMENTATION OF OBJ-T: A LANGUAGE FOR WRITING AND TESTING ABSTRACT ALGEBRAIC PROGRAM SPECIFICATIONS, Phd Thesis, University of California, Los Angeles, Ca., 1981

DOUBLE BIND AND EFFECTIVE COGNITIVE COUPLING IN HUMAN - MACHINE INTERACTION

Giovan Francesco Lanzara - Università di Bari

Graziella Tonfoni - Università di Bologna

RIASSUNTO

L'interazione tra utenti e sistemi manifesta in più casi problemi e patologie nel comportamento comunicativo dei primi. Un fenomeno di rilievo è costituito dal double-bind che può compromettere l'interazione tra utente e sistema. Questo articolo analizza la natura di tali problemi. Si esamineranno a questo proposito i livelli di incongruenza e dissimmetria fra l'utente reale e il modello di utente contenuto nel sistema.

ABSTRACT

Interaction between humans and machines exhibits communication pathologies such double-binds that may strongly affect the joint cognitive system performance. In this paper we explore the nature of the interactions and pathologies, we build a conceptual model of the human-machine double bind, and we point at some design proposals for reframing the interaction. We argue that a main source or double binds is the incongruity between the designer's currently practiced theories and assumptions and the premises and theories governing human's behavior as a problem-solver and communicative agent.

1. THE PROBLEM

Observation of interactions between humans and machines reveals some pathologies affecting human-machine communication. This is often characterized either by an implicity challenging interaction or by a sense of uneasiness and frustration by the user. As a consequence, human problem-solving and decision-making behavior and the overall cognitive performance of the human-machine cognitive system may be negatively affected. Among the factors accounting for the pathologies are the rigidity of the machine component and the implicit user's model designer creates and implements in building the machine and the interface [6], [5].

Observation of pathologies in human-machine interaction points out some incongruity between system design and the user's cognitive behavior. Currently practiced user's models seem to be incompatible or inconsistent with the set of assumptions, norms, strategies and expectations governing the user's effective performance and supporting his personal competence and commitment in a problem-solving or decision-making process. Such incongruity makes user's models sometimes difficult to be implemented, generating system's dysfunctional responses and dilemmas for the user.

Our research project is aimed toward:

(i) exploring the nature of the observed interactions and pathologies;

(ii) building a conceptual model of the pathology, which is a model of the double bind[1] between cognitive

This paper has been written when G.F. Lanzara and G. Tonfoni were Visiting Scholars respectively at MIT and Harvard University, Cambridge, MA. (USA). Research was partially supported by FORMEZ and CNR.

(1) Double bind relationship: a deep relationship between two communicative partners which influences communicative interaction in the following way:
a. a message is sent that (a) it makes some assertion; (b) it makes some assertion on its own assertion; (c) the two assertions are mutually incompatible;
b. the receiver of such message connot get out of the context carried by the message itself; he is neither allowed to refuse such message by linguistically criticizing (metacommunicating), nor able just to escape from it.
Any double bind relationship generates inherently paradoxical messages. Some examples double bind message are the following:
Be yourself!
Be spontaneous!
Such expressions contain an implicit cotracdition. Double bind relationship and messages have been deeply explored by the literature on pathologies and schizofrenia.

systems performing parallel problem-solving processes;

(iii) making some proposals for reframing system design, in order to produce a more effective cognitive coupling between human and machine and to prevent the occurrence of double bind phenomena.

Any AI system is not just a tool to be used but a consulting partner, which can reason and make decisions within a specific knowledge domain requiring expertise. Our basic assumption is that when human and machine interact they form joint cognitive system consisting of two partners performing a joint cognitive task [8]. The performance of such system is specific with respect either to the performance of the Human Cognitive System (HCS) or to the performance of the Machine Cognitive System (MCS). Communication pathologies like double binds may only arise when two fully cognitive systems engage in a communicative encounter [3], [7], [2].

Come out of an effective cognitive coupling between HCS acend MCS [8].

2. JOINT HUMAN-MACHINE COGNITIVE SYSTEMS: TYPES OF INTERACTIONS AND PATHOLOGIES

We may roughly distinguish between two basic models of HCS which designers refer to:

A. temptative-procedural HCS

B. optimalized[2] HCS

and between two basic of MCS:

C. consultant-advisory MCS

D. expert MCS

By coupling the two HCS models with the two MCS models we obtain three different types of human-machine interaction we exclude B - C interaction because no coupling reasonably exists between optimalized HCS and consultant-advisory MCS, because a consultant-advisory MCS presupposes a naive user, who is rather an expert in the field nor a student trying to acquire knowledge systematically; he rather has to be supported progressivaly through an easy, step by step interaction with the system.

A - C interaction

A - C interaction is typical in advisory systems[3]. As being the softest and most flexible ones, they are user-friendly and tend to generate experimental trial-and-error environments. MCS has a high adjustment level, and supports and coaches HCS in the problem-solving process. Interaction typically occurs by means of query/answer procedures. MCS does not fire one-shot solutions on specific knowledge domains, but rather takes advisory roles, providing informative support to HCS. MCS is an automated support to HCS to define problems and questions, organize information, generate plans, and suggest plausible answers. It activates evaluation, choice and balancing processes by HCS.

However, even though MCS does not replace HCS problem-solving role and skills, dependency can be generated for different reasons. For instance, HCS might not know where MCS is going to lead him, therefore he will tend to accept the reasoning procedure provided or somehow imposed by MCS. If HCS is not given the chance to track back the MCS problem-solving procedure and to understand where the procedure comes from, then, as a consequence, he might feel inadequate or out control, assuming a dependent and submissive behavior.

Another source of pathology in A - C interaction is due to HCS misrepresentation of MCS: HCS has misplaced expectations on what MCS should do or on who should do what. HCS expects MCS to be a problem-solver outputting correct and clearcut solution, and might get disappointed because MCS "doesn't make any decision". HCS then interprets MCS performance as incomplete, inadequate, or unsatisfatory. He denies himself to a process evaluator and would rather ba an executor, taking a "low profile" responsability attitude.

B - D interaction

Let us consider B - D intercation type in expert systems[4]. As being systems working on specific knowledge domains, they tend to generate "using" environments, where MCS has a low adjustment level and HCS is supposed to adapt to the machine's high level of performance. Since high performance standars are required to the optimalized HCS, great pressure is put on him, producing stresses in the interaction. MCS constantly challenges HCS, who is given two chances: he either accepts the challenge and competes with MCS, trying to hit the required performance targets (or even to be a better perfomer than MCS), or simply "given up", switching to submissive behavior.

MCS takes a problem-solving role on restricted domains: it is something which is called in to solve a problem for the human on the assumption that the problem is beyond the skill of the human. In B - D interaction MCS tends to replace HCS, whe may soon shift into submissive beha-

(2) Optimalized: an "optimalized system" is that kind of system which has been modelled in order to make any decision-making process the most efficient one in terms of low costs, high level of performance and short time effort.

(3) Advisory Systems: Advisory Systems are systems which are intended to be able to interact friendly, on a natural language conversation basis, with naive users in order to give advise in some very specific domains of expertise. The most signficant one have been produced at Cognitive Systems Co. by the research group of Prof. Roger Schank (Yale Univ., New Haven, CT, USA).

(4) Expert Systems: Expert Systems are systems which contain organized knowledge in some specific knowledge domain. They are not necessarily intended to be friendly; they are rather supposed to be an advise and support tool mostly for experts in the field. The most relevant examples are DENDRAL (in the field of Chemistry) or MYCIN (in the field Medical Knowledge). They are supposed to incorporate expert knowledge in a specific domain and to give advice in decision making.

vior, looking at himself just as an excutor of a MCS-determined process.

This kind of interaction is a strong generator of behavioral syndromes like disappointment, dependency, submissiveness, competitiveness, anger, and mistrust. An important source of pathology might be called the black box effect. HCS is put in a "take it or leave it" position as he confronts MCS output; he is basically left with no explanation of what MCS has done.

A - D interaction

ICAI systems[5] involve an interaction between a temptative-procedural HCS and an expert MCS. As being systems which allow HCS to penetrate and check the logic of MCS, they generate "learning" environments and support intelligent inquiry on problem-setting, plan generation, question relevance, analysis of counterfactual reasoning, plus trial-and-error problem-solving search. These systems embed a different model of human-machine interaction, based on the integration of different kinds of skill and knowledge. They detour the pathologies of the other types of interaction, but might generate another kind of pathology: being MCS completely "open" to HCS, the latter might use it for selfvalidating purposes, leading it to support whatever he wants to do, irrespective of whether that is right or wrong.

3. THE PHENOMENOLOGY OF HUMAN-MACHINE DOUBLE BIND

On a different degree all types of joint human-machine cognitive systems may exhibit pathologies in any kind of interaction. When HCS interacts with MCS outputting fixed and usually correct solutions, he either accepts the solution, abdicating his decision-making responsability, or rejects the solution, taking control on the machine and eventually trying to find or make up grounds to support his rejection. In the first case HCS "gives up", letting the machine do all the work and make the decision; by acting that way he might make the mistake of adopting the wrong solution. In tha second case HCS "takes over", filtering the output and eventually overriding the machine, but then he might make a mistake as well, by rejecting the right solution.

If HCS keeps accepting MCS-determined solutions, then he might end up feeling uneasy and diseppointed because he experiences MCS as dominant and his own problem-solving role and skills of little relevance in the process. Human does not feel dignified as a problem-solver. He feels uneasy with the machine because the machine is taking over authority and cognitive skills in a domain where HCS is supposed to be competent, and uneasy with himself because he feels inadequate or incompetent as a problem-solver. As sense of incompetence and loss of commitment increase, further dependency, submissiveness and disappointment are generated.

If HCS keeps disconfirming and rejecting MCS-solutions, and tries to find better ones himself, then he also might end up feeling uneasy, because MCS will be experienced as unreliable and bound to failure. He then becomes uneasy with the machine because the machine seems to be cognitively redundant and not providing much help after all, and uneasy with himself because, by becoming exceedingly manipulative and troublemaking, he loses criteria for performance evaluation. Moreover, engages in redundant problem-solving effort, which of course is time and cost consuming, with undue cognitive resources. As perception of MCS unreliability increases, further disappointment, competitiveness and mistrust are generated.

Two self-reiforcing loops are generated in the interaction. Submissive behavior will lead HCS to experience MCS as even more dominating and out of control, thus generating more submissiveness; on the other side, competitive behavior will lead HCS to redundancy of cognitive work. Dominant HCS behavior will generate more disconfirming attitudes, more uncertainty in the interaction and, in turn, more mistrust and competitive behavior (fig. 1).

By disconfirming the machine, HCS reiforces his own feeling of control, but as a side effect HCS tends to lose stable criteria for effectiveness. HCS has to constantly show that MCS fails and that he is a better performer, and in order to prove MCS unreliability he must keep disconfirming it. But if MCS fails too often, then there is no point in spending so much time in endless tests and computations. Competitive interaction generates cognitive redundancy, because HCS tends to duplicate work and time dedicated to problem-solving, thus outplaying the machine's role.

Observation of human-machine interaction has also shown that HCS may switch from a "giving up" to a "taking over" loop, and viceversa (fig. 1). If human feels too passive, or even threatened, he gets angry both at himself and the machine, and may suddenly switch to competitive behavior. Conversely, human might easily get fed up with his own redundant cognitive workload and switch to a more passive behavior. Thus, threatened by contradictory requirements, human takes systematically inconsistent behaviors to face and curb ambiguity. He keeps perpetuating submissive or competitive behavior, or oscillates between the two of them. He might also try to quit, but he is forbidden to do that, because he is "locked in" with MCS.

Either choice leads to dysfunctional responses, like feeling of personal incompetence, loss of involvement, unwarranted disconfirming attitudes, redundant cognitive effort, dismissal of the machine as "unreliable" or even 'useless", and, finally, loss of stable and clear criteria for effective-

(5) ICAI Systems: Intelligent Computer-Aided Instructions Systems are support tools in tutoring and teaching. They incorporate a set of questions in some specific knowledge domain which should be answered by a User/Student, who should also be able to ask the system for more information, if needed, and to check and keep control on his own knowledge acquisition process.
Some relevant ICAI Systems are SCHOLAR, WHY, WHUMPUS and SOPHIE which have been produced at Bolt Beranek and Newman Laboratories, Cambridge Mass., USA.

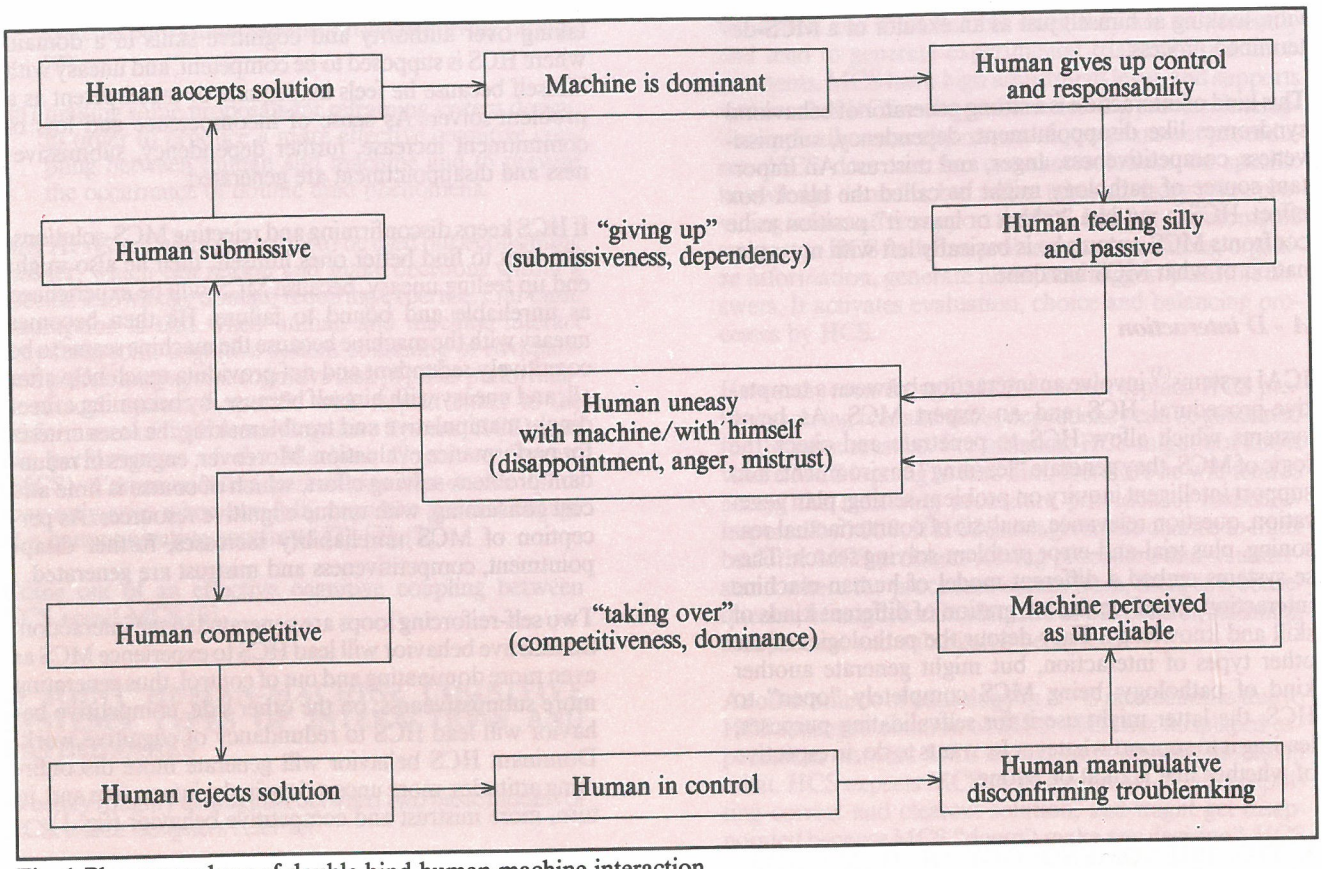


Fig. 1 Phenomenology of double bind human-machine interaction

ness and evaluation.

Such kind of double binds may dramatically lower the performance if the joint human-machine cognitive system. Decrease in performance is generally associated to fixation or perseveration effects in HCS's assessment of the state of some process, and with narrowing evaluation of alternative hypotheses, or else with excess of selective filtering and with too much broadening of the range of possible solutions [8].

4. THE USER'S DILEMMA AND THE DESIGN PROBLEM: A DIAGNOSIS

Even though there is no primary cause in a double bind situation, but a network of interlocking interactions, we argue that important sources of the user's dilemma are to be found in the incongruity between the design criteria embedded in the human-machine system and the theory governing user's behavior toward the machine [1].

First, let us consider the sharing of responsability issue: control on the problem-solving procedure is with MCS, but responsability for the choice of the solution (acceptance/rejection) is with HCS. HCS and MCS are designed in such way that control over the process is split: MCS controls the computing procedure, while HCS is responsible for solution and failures. Secondly, there is uncertanty over criteria for performance evaluation. If MCS performance is the central issue, then the target for the designer is to build better automated problem-solvers and get the user to behave accordingly: if on the contrary HCS performance is what gets evaluated, then control on the process should be placed with HCS, not with MCS. In fact it is quite inconsistent to take control away from HCS and hold him responsible for the solution and errors. Finally, if the joint system performance is what really matters, then the design target should be effective cognitive coupling and good communication between the two cognitive systems, i.e. a different kind of interaction.

HCS and MCS performances might be at odds with each other: the interaction of the two might yield as a result a low joint performance. The very design criteria oriented to maximize MCS performance and to achieve quality and correctedness of solution might undermine HCS confidence in the solution and make it difficult for him to be an effective performer. Ambiguity in design criteria originates from specific assumptions embedded in the user's model applied by the designer, i.e. in the designer's theory about user's behavior and human-machine interaction. Assumptions are not necessarily explicit: the designer can be weakly aware of them. Similarly, the user's responses to the situation are a conseguence of premises and theories the user holds about himself as a problem-solver and as a communicative agent, which may themselves be self-contradictory or incompatible with the designer's premises and theorie [1].

Having to face built-in contradictions in his own behavior

and in human-machine interaction and being exposed to contradictory injunctions by the designer, the user tends to give inappropriate behavioral responses that perpetuate the conditions generating dysfunctions and dilemmas.

The diagnosis performed above points to a global reframing of human-machine interaction, that involves the design of a different knowledge organization in MCS and of a new structure of communication between HCS and MCS.

5. REFRAMING THE DESIGN OF HUMAN-MACHINE COGNITIVE INTERACTION

The main problem in HCS-MCS interaction we are addressing is related to a reframing of MCS design, such that MCS can be viewed and interpreted by HCS as a decision-making problem-solving aid. In order to have effective interaction between HCS and MCS, which means cognitive coupling between two cognitive systems, MCS design must incorporate a model of its own reasoning procedures, which must be made explicit and have to be transparent to HCS.

Any HCS-MCS cognitive interaction takes place within a specific environment, under specific constraints. We can consider HCS as being an interface between MCS and a specific environment: by following such perspective we'll actually get a model of HCS as a context-sensitive data gatherer which will facilitate MCS data processing by choosing the most appropriate input in order to get an optimal output. Therefore the natural result of HCS - MCS interaction turns out to be MCS control on the decision-making problem-solving process (DMPS).

A different way of looking at HCS-MCS interaction is to consider MCS as a tool for making explicit and transparent any DMPS process to HCS by actually putting HCS in charge of a final decision about accepting or refusing MCS advice. MCS is a tool incorporating an explicit model of DMPS process and of the why and how such process has been selected and performed by MCS.

Any DMPS process consists on some kind of "Complex Reasoning" based on the interplay among the following reasoning modes:

— Domain Dependent Knowledge Expert Reasoning (DDKER), which means skilled and expert reasoning within a specific Knowledge Domain;

— Common Sense Reasoning (CSR), which is not strictly related to any specific Knowledge Domain, but is part of everyday life experience in the physical and social world;

— Learning by Experience Reasoning (LER), which consists on analyzing new cases in a specific Knowledge Domain, by comparing them to previous ones using recently acquired knowledge within a certain domain to modify and develop previously acquired skills.

Whereas DDKER can be presently incorporated within MCS, it is far more difficult to have CSR and LER actually work within MCS. Any MCS is to be viewed by HCS as acting on a DDKER level and only on a reduced scale on a CSR or LER level, given the actual status of research. An effective interaction is needed among different thinking procedures during DMPS processes, where HCS role consists on activating-coordinating-selecting and evaluating such procedures. Pathological attitudes by HCS arise when locus of control is expected to be part of MCS. Any Cognitive Step (which means progressive and sequentially ordered steps in the explicitation of the reasoning processes) performed by MCS requires a parallel Cognitive State by HCS. Such State does not necessarily correspond to the MCS Step, being MCS a model of certain kind of reasoning and not of the overall optimalized reasoning within a specific domain. No optimalized reasoning in a certain domain can exist in DMPS on a "a priori" level, it rather exists only if specific constraints and goals have been previously set by HCS. DMPS output by MCS has to be coherent given a certain premise and a set of goals, therefore subject to change whenever HCS decides to take control on the overall procedure. HCS model implied by MCS can be the result of a Target Determined Ahead Modelling (TDAM) aimed toward a pre-determined and idealized model how HCS should be and possibly act or finally become. In such case MCS incorporates a set of predetermined goals which are supposed to be reached by HCS after a cognitive interaction.

Pathological attitudes by MCS arise when a high level of adjustment is required in order to have Cognitive States exploited by HCS correspond to and match the respective Cognitive Steps exploited by MCS. MCS must incorporate an Explicit model of HCS such that HCS can decide whether or not to conform to such model or what to change about it, depending on his own needs and on the constraints of the specific situation where such interaction is actually taking place. Therefore MCS will have to incorporate an Interpretive Progressive Model of HCS such that to each Cognitive Step exploited by MCS a whole set of Plausible Cognitive States can be foreseen and handled.

Making HCS model explicit to the actual HCS implies the following design features in MCS:

— Dynamic vs Static Modelling: which means that MCS has to be flexible, therefore can be modified at each Step by HCS, who may need to stop or repeat some parts of the Step by Step process in order to give a good evaluation and get effective support during the DMPS process.

— Predictive vs Deterministic Modelling: which means that MCS is supposed to predict possible reactions and not to predetermine necessary responses by HCS, who may actually react in different and not always expected ways.

Explicit and Interpretative Progressive Modelling have to be considered as substantial features in the design refra-

ming of MCS, because they facilitate cooperation, interaction and actual coupling between HCS and MCS, by stressing the choice and control role exploited by HCS and the challenging-checking-supportive role of MCS. Leaving choice and control on HCS part does not imply cognitive redundancy, in other terms it does not necessarily produce multiple-conflicting or multiple-parallel reasoning procedures.

It might as well be the case that HCS and MCS develop identical or partially identical, conflicting or partially conflicting reasoning procedures during a DMPS process. We'll call it Multiple Strategies Reasoning Process (MSRP). MSRP has to lead to corresponding control by HCS on the DMPS process exploited by both MCS and himself in order to check the effectiveness and the coherence of the processes given a certain premise and a certain set of goals. Joint coupling consists precisely on actualizing the potential of a joint effort by HCS and MCS, therefore making explicit by MCS the analysis, evaluation and selection by HCS of the results of a set of reasoning processes which have been activated and exploited by HCS and MCS jointly or separately. A joint coupling is aimed toward MSRP evaluation more than just goal achievement, being this last one the result of context-sensitive evaluation and selection by HCS given MSRP activation.

An effective HCS-MCS cognitive interaction exists when HCS is aware of:

— actual lack of CSR and LER in MCS;

— relevance and consistency of DDKER in MCS;

— difference and possible mismatch between Cognitive Steps in MCS and Cognitive States by HCS;

— possible fixation or inconsistency in DMPS process by MCS, where HCS is not being locus of control;

— need of consistent and context-sensitive updating of MCS by HCS;

— possible interaction between MCS and multiple HCS, meaning more individuals activating different reasoning procedures during the same DMPS process;

— multiple and conflicting interpretations of data relevance and possible mismatch between data gathering and data processing by HCS and MCS;

— different "nature" of the cognitive organization incorporated by HCS (which is a human context-sensitive learning system) and MCS (which is an artificial model-determined and program-driven system);

— different degrees of expertise exploited by HCS during HCS-MCS interaction (novices, experts, partial experts), given the different nature and source of their respective expertise.

Being not aware enough of such constitutive problems within HCS-MCS cognitive interaction can cause breakdowns in effective communication and the insurgence of double bind phenomena, as we have illustrated un this paper. Misleading targets and constraint underevaluation are likely to produce pathological behaviors and attitudes by HCS based on false expectations, misinterpretation and low self-esteem, while not secceeding in goal achievement by interacting with MCS.

6. REFERENCES

[1] C. Argyris, D.A. Schon, THEORY IN PRACTICE: INCREASING PROFESSIONAL EFFECTIVENESS, San Francisco, Jossey-Bass., 1974

[2] G. Bateson, STEPS TO AN ECOLOGY OF MIND, New York, Chandler Plub. Co, 1972

[3] G. Bateson, D.D. Jackson, J. Haley, J.H. Weakland, TOWARDS A THEORY OF SCHIZOFRENIA, Behavioral Science, pp. 251-264, 1956

[4] M.J. Fitter, M.E. Sime, RESPONSIBILITY AND SHARED DECISIONMAKING, in H.T. Smith & T.R.G. Green (eds), "Human Interaction with Computers", Academic Press, London, 1980

[5] J. Kammersgaard, ON MODELS AND THEIR ROLE IN THE USE OF COMPUTERS, paper presented at the Working Conference on Development and Use of Computer-Based Systems and Tools, Aarhus, Denmark, august 19-23, 1985

[6] W.M. Newmann, R.F. Sproull, PRINCIPLES AF INTERACTIVE COMPUTER GRAPHICS, London, 2nd edition, McGraw-Hill, 1979

[7] P. Watzlawick, J. Helmick-Beavin, D.D. Jackson, PRAGMATICS OF HUMAN COMMUNICATION: A STUDY OF INTERNACTIONAL PATTERNS, PATHOLOGIES, AND PARADOXES, Norton and Co, New York, 1967

[8] D. Woods, COGNITIVE TECHNOLOGIES: THE DESIGN OF JOINT HUMAN-MACHINE COGNITIVE SYSTEMS, The AI Magazine, 6, 4, winter, pp. 86-92, 1986

LINGUAGGIO I:
UN SISTEMA PER LA RAPPRESENTAZIONE DELLA CONOSCENZA IN LINGUAGGIO NATURALE

A. Fabris, M. Cimini

Dipartimento di Scienze dell'Informazione

Università di Milano

RIASSUNTO

Obiettivo del lavoro è la definizione di un linguaggio di uso generale per la rappresentazione e la manipolazione della conoscenza che goda di proprietà di leggibilità vicine a quelle del linguaggio naturale senza condividerne le ambiguità che impediscono interpretazioni computazionali precise.
Le caratteristiche del linguaggio I (da Italiano) consistono in una sintassi semplice e naturale per costrutti che permettano di rappresentare conoscenze sia logiche che procedurali. A tal fine si sono scelte strutture linguistiche che uniscono tecniche di programmazione tipiche dell'Intelligenza Artificiale a quelle convenzionali.
Il linguaggio I, infatti, permette la programmazione basata su regole, strutture e chiamata di procedure.
L'elemento di base è la proposizione, che relaziona il soggetto, il verbo, e opzionalmente l'oggetto ed eventuali modificatori.
La realizzazione del lavoro si avvale di un sistema di interfaccia in linguaggio naturale.

ABSTRACT

Our principal goal is to develop a general-purpose language for knowledge representation with properties of readability near to natural language and without ambiguities that prevent precise computational interpretation.
Language I (from Italiano) embodies a simple and natural syntax for the representation of both logical and procedural constructs.
To this end language structures were chosen which provide a combination of both Artificial Intelligence and conventional programming features.
Every problem-solving system must have ways to represent its knowledge about the task domain. Such knowledge may include taxonomies of classes and their relationships and facts about objects and their relationships.
The basic element of representation is the proposition, which is a simple sentence that relates a subjects, predicate and optional object with optional prepositional modifiers.
Language I allows the user to store sentences in the database and to test the current contents of the database by a variety of condition forms.
To advance the state of a problem-solving system inference rules are applied (state-driven and goal-driven inference).
The system uses a natural language interface.

1. INTRODUZIONE

Ogni sistema deve scegliere un modo di rappresentare la conoscenza. In particolare il linguaggio I consente, analogamente a Prolog di rappresentare la conoscenza mediante implicazioni:

se <ipotesi> allora <tesi>

Tale tecnica è stata scelta poiché, data la sua intrinseca modularità, facilita l'inserimento incrementale di nuove conoscenze.

La possibilità di combinare sentenze rappresentanti relazioni mediante operazioni logiche consente inoltre, nello stile della programmazione logica, una facile formulazione di una vasta classe di problemi.

Il primo passo compiuto è stato l'individuazione di un sottoinsieme della linga italiana minimale e, nella misura del possibile, completo in rapporto alla capacità di rappresentazione della conoscenza.

Il modello di rappresentazione della conoscenza seguito è quello logico.

Per trattare in modo automatico le informazioni dob-

biamo esprimerle in un linguaggio non ambiguo. Il linguaggio della logica simbolica è sufficientemente preciso per essere compreso e manipolato in modo automatico e può essere considerato, inoltre, una forma semplificata del linguaggio naturale e del ragionamento umano.

La logica simbolica ha una duplice applicazione: in analisi linguistica come strumento per la rappresentazione delle sentenze e delle regole sintattiche che la governano e in programmazione per la formulazione e la risoluzione dei problemi. In particolare permette la connessione di sentenze di base e l'esecuzione di istruzioni condizionali.

Per modificare o dedurre delle informazioni è stata fornita la possibilità di applicare due tipi di inferenza: goal-driven e state-driven.

Il sistema consente anche la programmazione di tipo procedurale: possono essere definite ed eseguite recursivamente funzioni, procedure ed espressioni aritmetiche.

2. LINGUAGGIO NATURALE E LOGICA

In ogni sentenza del linguaggio naturale il verbo esprime una relazione tra individui.

— Luca studia inglese.

Sentenze più complesse esprimono che una condizione atomica implica una conclusione atomica:

— se Y è un uomo allora Y è mortale.

Dove Y è una variabile che indica un qualsiasi individuo. Le sentenze possono avere molteplici condizioni e conclusioni (in alternativa o in congiunzione) per esempio:

X è un rettangolo
se X ha quattro lati
e X ha quattro angoli retti.

Tali sentenze sono chiamate clausole.

In generale una clausola esprime che un insieme di condizioni implica una conclusione. Condizioni e conclusioni esprimono relazioni tra individui. Essi possono essere fissati, perciò chiamati simboli costanti, e indicati mediante parole del tipo: Luca, inglese oppure possono essere arbitrari, chiamati variabili, come pure per esempio: X, Y. Le variabili in clausole differenti sono distinte anche se hanno lo stesso nome. Il nome della variabile ha significato solo nel contesto della clausole in cui essa si trova.

In quanto precede è facile riconoscere una descrizione non formale della logica a clausole. È conveniente esprimere le formule atomiche, che servono come condizioni e conclusioni di clausole, in una forma semplificata anche se meno naturale. Il nome della relazione è scritto di fronte alla formula atomica, seguito dalla sequenza di nomi di individui a cui la relazione è applicata.

I simboli di predicati binari invece di scriverli tra i loro argomenti si scrivono in forma prefissa cioè:

X è il padre di Y

diventa

padre (X, Y).

In modo analogo si comportano i simboli di predicati unari e di preicati con un numero di argomenti maggiore di due ossia:

X è buono

diventa

buono (X).

Ci sono molte occasioni in cui la logica usa una variabile, ma l'Italiano usa una parola che si riferisce a un tipo specifico (o ad una classificazione) di individui. È tipico della logica indicare i tipi usando un simbolo di predicato unario. Così la sentenza italiana

tutti gli uomini sono mortali

potrebbe essere espressa mediante la clausola

essere_uomo (X) <——— essere_mortale (X)

3. LINGUAGGIO I: UN SOTTOINSIEME DEL LINGUAGGIO NATURALE PER LA RAPPRESENTAZIONE DELLA CONOSCENZA

Ogni sistema deve scegliere un modo per rappresentare la conoscenza.

La conoscenza in generale è data mediante tassonomia fra classi, fatti applicati ad oggetti e loro relazioni.

Il modello di rappresentazione da noi seguito, come abbiamo visto, è quello logico. Nel seguito ci proponiamo di associare ad ogni sentenza italiana una o più clausole Prolog.

In Italiano l'elemento fondamentale della rappresentazione è la sentenza. La sentenza mette in relazione il soggetto, il predicato verbale e opzionalmente l'oggetto con eventuali modificatori preposizionali.

Il linguaggio I permette 5 tipi di preposizioni base o sentenze primitive:

— appartenenza a classi
— sentenze predicative
— forme verbali transitive e intransitive
— frasi preposizionali

Le sentenze primitive vengono trattate mediante rela-

zione N-arie. Una relazione N-aria è del tipo:

nome-relazione(X(1),X(2),...,X(N)).

Per esempio nella relazione ternaria

vendere__a(Mario,automobili,Milano).

"vendere__a" rappresenta la relazione tra un venditore, il prodotto che deve essre venduto e la città di vendita.

Le relazioni N-arie forniscono un'interpretazione di sentenze dichiarative italiane.

3.1 Elementi di base del linguaggio

Gli elementi di base del linguaggio I sono: l'insieme dei caratteri, i tipi di dati primitivi e le parole riservate o parole chiave.

Il sistema accetta il seguente insieme di caratteri:

spazio CR " ' ( ) ☆ + — /
0 1 2 3 4 5 6 7 8 9 : ; > = < ?
A B C ... Z a b c ... z

Ogni sequenza di caratteri compresa tra due caratteri separatori è detta parola.

I caratteri terminali, . e ?, indicano la fine di una regola o di una azione o di una domanda.

Gli atomi sono i tipi di dati primitivi del linguaggio I. Ci sono diversi tipi di atomi disponibili, ognuno dei quali rappresenta una diversa informazione:

— nomi

Giovanni, Africa

— stringhe

"questa è una stringa", "Parser in Prolog"

— costanti numeriche

1986, 1, ...

— variabili

X, Y ..

— parole chiave

ricorda	cancella	se
allora	altrimenti	calcolare
esegui		

— operatori o simboli d'interpunzione

+ , — , > , < , . , ? , : , . . .

3.2 Appartenenza a classi

Il verbo ausiliare "essere" ha interpretazioni diverse a seconda che sia unito ad un aggettivo o a un nome. Tali sentenze esprimono la relazione di appartenenza di un elemento ad un insieme.

Esempio:

— Maria è una maestra (significa che Maria appartiene all'insieme delle maestre).

— Giorgio è un commerciante (significa che Giorgio appartiene all'insieme dei commercianti).

Ci potrebbero essere altri elementi che appartengono alla medesima classe.

Esempio:

>> Maria è una maestra.

è rappresentato mediante la relazione:

essere__maestra(Maria).

Evidentemente il predicato ha come argomento l'elemento della classe.

3.3 Sentenze predicative

Le sentenze predicative definiscono proprietà di un elemento di un insieme.

Esempio:

Marco è felice.
Bruno è un eccentrico.

L'aggettivo è trattato come un predicato il cui argomento è il soggetto della sentenza. Le sentenze predicative sono interpretate come relazioni unarie. La seconda sentenza verrà trasformata in:

eccentrico(Bruno).

Questo per differenziare la traduzione della sentenza

Bruno è un eccentrico eccentrico.

che indica invece l'appartenenza dell'eccentrico Bruno alla classe degli eccentrici.

Sono accettate anche le sentenze che presentano più aggettivi dopo il verbo ausiliare "essere".

Esempio: data la sentenza

Marcella è intelligente, volenterosa e capace.

viene tradotta nei tre predicati:

intelligente (Marcella).
volenterosa (Marcella).
capace (Marcella).

Aggettivi e nomi di classi sono strettamente connessi, ma hanno una semantica molto differente.

Nella sentenza dove il verbo principale "essere" ha funzione di copula e quindi è unito ad un aggettivo abbiamo stabilito che l'aggettivo si applica all'elemento.

Esempio:

Marco è sapiente.
Marco è un sapiente.

La prima frase viene interpretata come un predicato con argomento Marco. Nel calcolo dei predicati potrebbe essere rappresentata mediante:

sapiente (Marco).

La seconda frase tratta la parola "sapiente" come un insieme. Marco appartiene a tale insieme. Si rappresenta mediante la relazione di appartenenza cioè:

essere__sapiente (Marco).

3.4 Forme verbali transitive e intransitive

Le voci verbali al modo indicativo e al tempo presente sono usate per creare altre relazioni. Relazioni unarie possono essere formate usando verbi di forma intransitiva, mentre relazioni binarie usando un verbo transitivo che richiede complemento oggetto.

Nelle relazioni unarie, il soggetto del verbo è l'unico argomento della relazione, mentre in quelle binarie il soggetto è il primo argomento e il complemento oggetto il secondo.

Esempio:

Lucia dorme ===> dormire (Lucia).
Ugo studia Inglese ===> studiare (Ugo, Inglese).

Ognuna delle sentenze primitive può essere seguita da una frase preposizionale. Una frase preposizionale specifica ulteriormente la sentenza in cui è inserita.

Da rilevare il fatto che la preposizione limita il predicato associato alla sentenza tradotta. Infatti la preposizione semplice è aggiunta, mediante underscore, alla forma infinita del verbo per formare il nome della relazione e i suoi argomenti sono il soggetto, l'eventuale complemento oggetto e il nome preposizionale.

Carlo lavora all'Honeywell.
lavorare__a (Carlo, Honeywell).

4. LE AZIONI

È possibile agire sulla base di conoscenza mediante azioni.

Le azioni modificano la base di conoscenze, permettono la costruzione di programmi, eseguono la verifica di condizioni.

Ci sono diversi tipi di azioni:

— sentenze condizionali

— procedure e funzioni

— inserimento e cancellazione di fatti e regole.

4.1 Condizioni

Le condizioni sono composte da una o più sentenze primitive. Ogni proposizione di base controlla una relazione o un predicato. La conclusione ottenuta è usata come risultato del test. Una condizione è vera se tutte le limitazioni verificate sulla base di conoscenze sono vere, altrimenti è falsa.

Le sentenze, che costituiscono una condizione, possono essere combinate usando i connettori logici "e" e "oppure" per creare predicati logici composti.

Per esempio:

se X è il padre di Y e X è il padre di Z
allora Y è il fratello di Z.

4.2 Definizione delle regole d'inferenza

La conoscenza asserzionale circa un particolare problema è separata in due categorie: regole e fatti.

Le regole consistono di quelle asserzioni date in forma implicazionale. Esse esprimeno conoscenza "generale" su un particolare dominio e sono usate come regole di produzione.

I fatti sono asserzioni non espresse come implicazioni. Essi rappresentano la conoscenza "specifica" su un particolare problema. Le regole modificano lo stato della base di conoscenze. Infatti permettono di dedurre delle nuove conoscenze mediante la loro applicazione.

Si distinguono due tipi di regole nel nostro lavoro. Dai linguaggi PLANNER-like abbiamo considerato regole d'inferenza goal-driven unitamente al concetto di base di conoscenza proposizionale. Da sistemi esperti, come MYCIN e RITA, abbiamo ricavato l'idea delle regole di tipo state-driven ("se condizione, azione").

Una regola goal-driven cerca di verificare una affermazione risalendo tutti possibili antecedenti logici finché si sono ottenute sufficienti informazioni per concludere la ricerca oppure non esistono alternative di ricerca. Per questo processo viene usato il motore inferenziale già esistente in Prolog.

Una regola state-driven agisce così: verifica l'antecedente di una regola rispetto la base di conoscenze e prosegue nell'esecuzione delle azioni richieste:

Esempio:

X è il nonno di Y
se Z è il padre di Y e
Z è il figlio di X.

Interpretazione:

```
essere_nonno (X, Y,) :—
        essere_padre (Z, Y),
        essere_figlio (Z, X).
```

4.3 Procedure e funzioni

Abbiamo fornito la possibilità di definire dei costrutti non solo di tipo dichiarativo, ma anche procedurale. L'utente può quindi definire una funzione o procedura mediante il costrutto "per calcolare". La procedura espressa in linguaggio I viene tradotta in Prolog e memorizzata nella base di conoscenze. tale procedura può essere richiamata usando l'azione "esegui".

Nella testa del costrutto si definisce il nome delle procedura o funzione e i relativi parametri di ingresso e di uscita. I parametri potranno essere sia delle variabili che delle costanti. Il corpo è una lista di azioni che verranno eseguite sequenzialmente.

Supponiamo di dover definire la funzione che calcola il fattoriale, scriveremo:

```
per calcolare il fattoriale X di Y
        se Y = 0 allora X = 1,
        altrimenti Z = Y — 1,
        esegui il fattoriale L di Z,
        X = L ☆ Z.
```

La rappresentazione interna di tale costrutto è:

```
fattoriale (Y, X) :—
        (Y = 0,
         X = 1;
         Z is Y — 1,
         fattoriale (Z, L),
         X is L ☆ Z).
```

Il costrutto viene tradotto come una clausola Prolog.

4.4 Azione di memorizzazione e cancellazione

Le azioni di inserimento e cancellazione sono usate per effettuare manipolazioni sulla base di conoscenze, cioè per incrementarne il contenuto.

La base di conoscenze contiene sentenze dichiarate. La proposizione che segue la parola chiave "ricorda:" viene interpretata come una sentenza. La sua traduzione viene memorizzata nella base di conoscenze.

Esempio:

ricorda: Marco è uno studente.

Il fatto:

essere_studente (Marco).

viene inserito nella base di conoscenze.

L'azione di cancellazione ha come argomenti una sentenza. L'azione di cancellazione ha come effetto l'eliminazione della relazione dalla base di conoscenze.

5. APPLICAZIONE DEL LINGUAGGIO I

L'idea che si vuole sviluppare in questo paragrafo è di accedere ad una base di conoscenze mediante domande poste in linguaggio I. Usando la sintassi del linguaggio si formulano domande sulle conoscenze memorizzate.

Un linguaggio per l'interrogazione di data-base o query language è uno strumento utile.

Un sistema d'interrogazione in linguaggio pseudonaturale elimina la necessità di imparare la sintassi del query language.

Il linguaggio processa una domanda in due passi:

$$\text{DOMANDA} \xrightarrow[(1)]{\text{comprensione}} \text{PROLOG} \xrightarrow[(2)]{\text{esecuzione}} \text{RISPOSTA}$$

Nel primo passo ci si chiede che cosa la domanda significa si cerca cioè di intuire il problema che deve essere risolto.

Nel passo successivo ci si chiede quale sia la risposta in rapporto alla domanda.

Per la definizione del sistema abbiamo connesso concetti di AI, di programmazione logica, utilizzando il linguaggio Prolog, e di gestione di un data-base. Per quanto concerne AI e programmazione logica usiamo il concetto di rappresentazione della conoscenza e il meccanismo della deduzione, mediante regole d'inferenza.

Sulla gestione di un data-base abbiamo utilizzato il concetto di data-base relazionale.

L'uso della logica per la descrizione dei dati elimina la distinzione tra programmi e basi di dati. Le strategie che si applicano per l'esecuzione di un programma sono le stesse che si applicano per reperire le risposte alle domande.

1) dal punto di vista del theorem-proving, un data-base può essere considerato come un sistema question-answering dove i fatti sono rappresentati da formule logiche; allora rispondere a una domanda, a partire dai fatti è come dimostrare che la formula corrispondente alla risposta è derivabile dalle formule che rappresentano i fatti; quindi la ricerca deduttiva diventa fondamentale.

2) se la domanda è scritta in un linguaggio d'interrogazione ad alto livello l'interpretazione di tale domanda corrisponde all'esecuzione di operazioni che si possono definire intelligenti. Il data-base può essere considerato una base di conoscenze.

3) poiché un data-base può essere considerato come un programma logico, il reperimento di una informazione avviene automaticamente: applicando il principio di risoluzione. Per esempio, definito un predicato per una relazione n-aria p, il reperimento dell'i-esimo argomento è ottenuto ponendo la domanda:

?— p(v(1), . . . ,v(i—1),X,v(i+1), . . . ,v(n)).

5.1 Esempio di applicazione

Consideriamo la domanda di tesi che gli studenti devono compilare per sostenere l'esame di laurea.

Ogni tipo di conoscenza asserzionale riguardante il particolare dominio può essere inserita mediante l'azione di memorizzazione fornita dal sistema:

[F1] Torquati è il relatore di Fabris.
[F2] Degli_Antoni è il correlatore di Fabris.
[F3] Torquati lavora presso la ditta Honeywell.
[F4] Fabris è una tesista.

Durante una qualsiasi sessione di lavoro possono essere inserite regole d'inferenza del tipo:

[F1] la ditta Y collabora con l'Università
se R lavora presso la ditta Y,
R è il relatore di X o
R è il correlatore di X e
X è un tesista.

Dati i fatti F1, F2, F3, F4 e applicando la regola R1 se viene posta la domanda:

la ditta Honeywell collabora con l'Università?

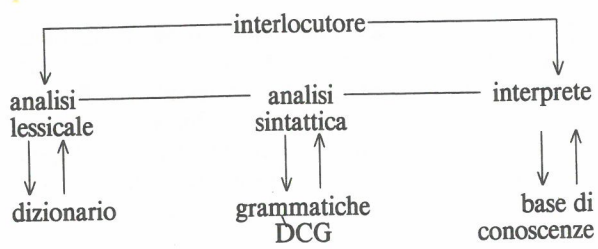
il sistema inferisce:

si, la ditta Honeywell collabora con l'Università.

Utilizzando le regole d'inferenza e i fatti inseriti nel data-base sarà possibile dedurre delle nuove informazioni.

6. SISTEMA D'INTERFACCIA

Il sistema la cui struttura è rappresentata in figura ê stato sviluppato interamente in Prolog.



Non avendo uno specifico campo di utilizzazione, il sistema è dotato di un lessico piuttosto ristretto ma facilmente espandibile rispetto a specifiche utilizzazioni. Riconosce oltre a parole chiave del linguaggio, il verbo essere e avere in forma integrale e nei tempi passato presente e futuro i verbi di movimento (andare, venire), di relazione, di conoscenza e di comunicazioni. Sono inoltre memorizzate tutte le preposizioni semplici e articolate, articoli determinativi e indeterminativi.

La frase inserita viene dapprima analizzata sotto il punto di vista lessicale verificando la validità ortografica delle parole inserite con una ricerca di esse nel dizionario del sistema. Eventuali errori di battitura possono essere corretti dallo spelling corrector in modo automatico.

Segue l'analisi sintattica della frase e la sua interpretazione.

6.1 Analizzatore lessicale

L'analizzatore lessicale isola e accetta i singoli vocaboli che costituiscono la frase in ingresso.

I nomi propri, le costanti numeriche e le variabili vengono passate direttamente all'interprete senza ulteriori controlli. Le altre parole vengono ricercate nel dizionario del sistema e ad ognuna di esse è associato uno o più records lessicali.

Esempio: data la sequenza di caratteri in input

le stelle sono luminose

otteniamo i records:

le → articolo, femminile, plurale.

stelle → nome, femminile, plurale.

sono → essere, presente ind., persona 1ª, singolare.

sono → essere, presente ind., persona 3ª, plurale.

luminose → aggettivo, femminile, plurale.

Alcuni vocaboli come nell'esempio "sono" possono essere associati a più record lessicali. In questo caso sarà l'analisi sintattica successiva a stabilire il record appropriato.

La maggior parte delle parole della lingua italiana è variabile cioè assume diverse flessioni a seconda del numero, del tempo e del genere.

La parte costante della parola è detta radice mentre quella variabile desinenza. L'idea di memorizzare in forma integrale tutte le flessioni di una parola o di un verbo risulta assai dispendioso per l'enorme spazio di memoria richiesto.

Nel linguaggio I radici e desinenze sono memorizzate separatamente.

Le desinenze risultano essere poche, rispetto le radici, essendo comuni a più gruppi di vocaboli. È evidente che non tutte le radici possono essere associate ad una qualsiasi desinenza. Così le desinenze sono raggruppate in base alla radice ad esse compatibile.

In questo modo è stato possibile isolare alcuni insiemi di desinenze che sono associabili a radici di verbi irregolari molto frequenti nella lingua italiana.

6.2 Analizzatore sintattico

L'analizzatore sintattico ha la funzione di verificare che la frase inserita sia corretta sintatticamente.

La sintassi del linguaggio I è interamente espressa in notazione DCG (Definite Clause Grammar).

Tale notazione utilizza un formalismo di tipo logico e gode di particolari caratteristiche di chiarezza e concisione. Una DCG è una CFG (Context Free Grammar) estesa in cui al posto dei normali simboli grammaticali figurano termini logici.

In una grammatica CFG le produzioni sono del tipo:

```
nt  ⟶  body
```

dove nt è un simbolo non-terminale del linguaggio e body è formato da una sequenza di simboli non-terminali o da un simbolo terminale.

Una frase può essere generata dalla produzione:

```
phrase  ⟶  noun_phrase,
           verbal_phrase
```

dove noun_phrase e verbal_phrase sono simboli non terminali definiti da altre regole CFG.

```
noun_phrase ⟶ article, noun

noun_phrase ⟶ noun
```

Infine la grammatica possiede produzioni del tipo:

```
noun     ⟶  [leone]

article  ⟶  [il]
```

dove gli elementi racchiusi tra parentesi sono i simboli terminali.

La CFG possono essere generalizzate nel seguente modo:

— i simboli terminali possono essere termini composti oltre ad atomi.

```
np(X,S)        sentence (S)
```

— nella parte destra di una regola possono esserci, oltre a simboli non terminali e liste di terminali, anche delle sequenze di chiamata a procedura scritte dentro parentesi graffe. Esse esprimono condizioni da soddisfare affinché la regola sia apllicata.

```
noun(N)  ⟶  [n], (is_noun(N))
```

Cioè N appartiene ai nomi se N esiste nel dizionario ed è un nome. Il dizionario è della forma:

```
is_noun(leone)

is_verb(mangia)
```

La notazione DCG fornisce lo strumento per rappresentare in modo semplice e conciso la dipendenza dal contesto. Gli argomenti dei simboli non-terminali possono essere utilizzati oltre che per costruire il parse tree anche per contenere informazioni riguardo a caratteristiche morfologiche della stringa analizzata. In questo modo vengono verificati accordi a breve e a lunga distanza nel genere, numero, persona.

La sintassi del inguaggio I è stata interamente espressa in notazione DCG.

Ad esempio la struttura <se> Condition <allora> Action è rappresentabile come:

```
statement_if(stat_if(cond,Act)):—
        [se],
        Lista_condizioni(Cond),
        [allora],
        Azioni(Action).
```

Dove la lista delle condizioni può essere costituita da una o più frasi italiane del tipo:

```
sentence(s(NP,VP))  ⟶  noun_phrase(NP,Person,
                       Number),
                       verbal_phrase(VP,
                       Person,Number).
```

unite mediante congiunzioni o disgiunzioni.

Mediante l'inserimento di opportune procedure di controllo sono stati gestiti accordi fonetici tra articolo e nome.

Esempio:

```
gli tuoi amici
```

non viene riconosciuta come parte nominale corretta poiché non c'è accordo morfologico tra articolo e nome.

Infatti:

```
noun_phrase(np(NP), Number, Type):—
        article(Art, Number,Type),
        noun(Noun,Number,Type),
        (agreement(Art,Noun,Type,Number)).
```

Le grammatiche DCG possono essere facilmente trasformate in clausole di Horn ed eseguite mediante la "proof procedure" implementata per il Prolog. La fase di parsing di una frase in input coincide con la verifica di un goal del tipo:

```
sentence(S,SO,sentence(T)).
```

L'input viene analizzato da sinistra a destra secondo la tecnica di parsing top-down left-right. In caso di successo la variabile "T" contiene la struttura sintattica associata all'input. Tale struttura rappresenta l'input dell'interprete del linguaggio I.

7. RINGRAZIAMENTI

Si ringrazia il dott. F. Torquati per il costante interessamento e i continui spunti offerti al lavoro di tesi e il Prof. G. Degli Antoni per la sua opera di supervisione e critica.

Il lavoro è stato sviluppato come tesi di laurea presso la Honeywell Information Systems Italia.

8. BIBLIOGRAFIA

[1] A.V. Aho, J.D. Ullman, THE THEORYOF PARSING TRASLATION AND COMPILING, Vol. I, Prentice Hall, Henglewood Cliffs, New Jersey, 1972

[2] W.F. Closkin, C.S. Mellish, PROGRAMMING IN PROLOG, Springer-Verlag, 1981

[3] Del Canto, UN MODELLO DI RAPPRESENTAZIONE DEL LESSICO PER LA ELABORAZIONE AUTOMATICA DELLA LINGUA ITALIANA, Rivista di informatica, Vol. XII, n. 4, Ottobre-Dicembre 1982

[4] Li Deyi, A PROLOG DATABASE SYSTEM, Research Studies Press Ltd., 1984

[5] F. Hayes-Roth, D. Gorlin, S. Rosenchein, H. Sowizral, D. Waterman, RATIONALE AND MOTIVATION OF ROSIE, Rand Co., Santa Monica, 1981

[6] F. Hayes-Roth, D. Gorlin, S. Rosenchein, H. Sowizral, D. Waterman, ROSIE LANGUAGE REFERENCE MANUAL, Rand Co., Santa Monica, 1983

[7] F. Hayes-Roth, D. Gorlin, S. Rosenchein, H. Sowizral, D. Waterman, PROGRAMMING IN ROSIE: AN INTRODUCTION BY MEANS OF EXAMPLES, Rand Co., Santa Monica, 1983

[8] R. Kowalski, LOGIC FOR PROBLEM SOLVING, Elsevier Science Publishing Co, 1985

[9] F. Pereira, LOGIC FOR NATURAL LANGUAGE ANALYSIS, Technical notes 275, SRI International, 1983

[10] F. Pereira, D.H. Warren, PARSING AS DEDUCTION, Artificial Intelligence Center, SRI International

[11] J. Peterson, COMPUTER PROGRAMS FOR SPELLING CORRECTOR, Springer Verlag, 1979

[12] S.A. Tarlund, K.L. Clarck, LOGIC PROGRAMMING, Academic Press Inc., 1984

[13] M. Wallace, COMMUNICATING WITH DATA BASE IN NATURAL LANGUAGE, Ellis Horwood limited, 1981

D.H. Warren, F.C.N. Pereira, DEFINITE CLAUSE FOR LANGUAGE ANALYSIS - A SURVEY OF THE FORMALISM AND A COMPARISION WITH AUGMENTED TRANSITION NETWORK, Artificial Intelligence, 1980

[14] T. Winograd, LANGUAGE AS A COGNITIVE PROCESS, Vol. 1, Syntax, Adison Weslay, 1983

GRAPHICS CODE GENERATOR BASED ON FORMAL SEMANTIC DESCRIPTION PRINCIPLES

Ivan Jelinek

Czech Technical University - Department of Computer Science

Praga

RIASSUNTO

L'articolo presenta un nuovo metodo per la costruzione automatica dei generatori di codici grafici. Il metodo è basato su un modello teoretico di traduzione e derivato dalla teoria delle semantiche denotazionali.

ABSTRACT

The paper presents a new method of automatic constrution of graphic code generators. The method is based on a theoretical model translation and derived from denotational semantics theory.

1. GRAPHIC CODE GENERATOR

Graphic code generator construction and implementation for different graphic processors and for different applications is a laborious work. It is clear that the construction and implementation of a whole new graphics system for a new application or for new configuration of peripheral graphic equipments is not economical. Therefore, there arises a need to standardize the common properties of the graphic systems independent of the application and use of graphic peripherals. The goal of this effort was the proposal of the standards known as CORE and GKS [1].

In every graphic system we can distinguish three layers [2]. The top layer accesible to the user is predetermined by the users applications: computer aided design, tutorial system etc. The other layers must be distinguished by the implementator - the *peripheral dependent layer* which depends in the peripheral equipment configuration and the *peripheral independent layer* which is provided by the graphical standard. The language means for the application layer description is called the graphic system language. The means for the peripheral independent layer and peripehral dependent layer is called the *graphic language* and the *graphic code* respectively.

From the point of view of the graphic system the graphic language acts as an interlanguage and the means which translates the program of the graphic language into graphic code is called the *graphic code generator.* The means of translation from the graphic system language into the graphic language can be realised as a certain type of processor. For the aim of this paper we call the graphic language - from the point of view of the graphic code generator - *an input language* (LI) and the graphic code an *output language* (LO) of the graphic code generator.

The construction and implementation of the graphic code generator is a key problem in the implementation of each graphic system. Therefore, the idea of automatic construction of the graphic code generator has appeared.

2. AUTOMATIC SYNTHESIS OF CODE GENERATOR

First, the possibility of using automatic synthesis methods for the programs construction was studied. But such general methods proved to be somewhat cumbersome and complicated. Therefore, these methods were abandoned and methods for the synthesis of the special programs - translators - were studied [3].

The basic principle of an automatic synthesis of translator

is a complete description of the source language and the object code. By complete description we understand the description of both the syntax and the semantics of a language; the pragmatics is not reasoned [4]. From these two description the translator generator generates the translator. But no such ideal translator generator exists [5].

The key problem of the automatic construction of the translator is the problem of choosing suitable means for the formal description of the syntax and the semantics of programming language. This means must be mathematically, precise, not complicated, suitable for realistic programming languages as well as this one must be computer processing oriented. Suitable means for syntax description exists and it is currently succesfully used for automatic construction of some translator parts [6].

For semantic description there are a lot of suitable means but semantic description means are denotational semantics [7] and attribute grammars [8]. In our approach we have used denotational semantics.

Denotational semantics is widely used and has often been described but the present author has not found a plublication in which a precise definition of denational semantics is available from the practical point of wiev. Therefore in our publication [9] the denotational semantics definition was formulated as a group of five elements:

$$DS = (G, SYD, SED, D, SR),$$

where
DS — denotational semantics,
G — context-free grammar,
SYD — syntactic trees domains,
SED — semantic domains,
D — semantic assignment,
SR — semantic equations.

3. FORMAL TRANSLATION DESCRIPTION

On the base of the denotational semantics definition was defined the function *vmean* mapping the sentences of the LI language into the *input language semantics domain (ILSD).* The element of the ILSD - wmi (wmi = *vmean* (wsi), wsi - the sentence (element) of LI) is called the *meaning of the sentence wsi in denational semantics DS.*

The output language semantics could described by means of the denotational semantics DS-but as output language LD (graphic code) is a simple language, the *simple type denotationa semantics (STDS)* was defined. The STDS is the simpler type of DS and for every STDS there exists a DS. For STDS the function *amean* was defined by mapping the sentences of the output language LO into the *output language semantics domain (OLSD).* The element of OLSD - wmo (wmo = *amean* (wso), wso - the sentence (element) of LO) is called the *meaning of the sentence wso in STDS.*

Prior top present the construction of the translator on the base of the denotational semantics we should say what mean by the term translation.

Translation is a function (called *comp*) which meets the following condition:

$$amean \circ comp = \delta \circ vmean \quad (*)$$

where

amean, vmean were described above,

δ — the auxiliary function mapping the elements of ILSD into OLSD; the funtion δ is for comparing the meaning of the elements of ILSD and OLSD,

comp — the defined translation function,

o — operator of function composition.

The fulfilling of condition (*) corresponds to the requirement of the commuting of semantics diagram (fig. 1):

Condition (*) can be a basis point for the proof of the correctness of some translation, if the functions *vmean, amean* an δ are known. We do not want to use the above condition for a translation proof but we use this condition for the construction of the translator from the knowledge of three functions *vmean, amean* and δ. The question is how.

4. GRAPHIC CODE GENERATOR SYNTHESIS

One way to solve the problem of the synthesis of transformation would be to construct the "inverse" function to *amean* function (see fig. 1). But there is no such "inverse" function - as "inverse" one would be ambiguous. Two different elements of LO - wx, wy - can exist for which:

amean (wx) = a and *amean* (wy) = a, a ϵ B1OLSD, that is the sentences wx, wy have the same meaning in OLSD.

But we need not distiquish the elements of the language LO corresponding to the meaning a (a ϵ OLSD). We are not interesting in concrete syntactic "form" of these

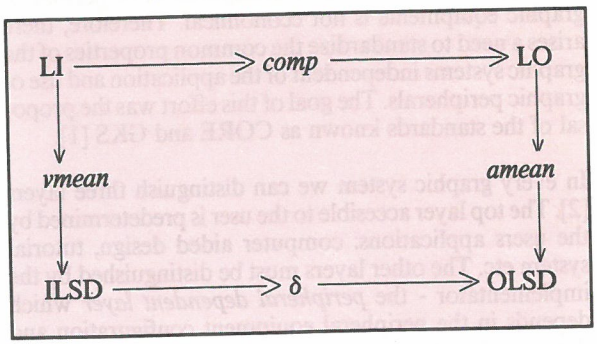


Fig. 1

elements - sentences of language LO. We can define the *factors* (of the sentences of LO) in wich every sentence (of different syntactic form) has the same meaning a (a ϵ OLSD). Therefore we can also define the *equivalence relation* onthe LO. This equivalence relation generates on the LO the *factor syntactic domain (FDCE) of the equivalent LO sentences* (see fig. 2).

Not all elements of the OLSD domain are meanings of some sentences of the LO. Therefore, we determine the part if the OLSD domain which contains those elements which are the meanings of all sentences of the LO. This part of the domain we call the *subdomain (SD) of the domain OLSD* - see fig. 3.

The new function *ameanf* (one to one) can be now defined - *ameanf* maps the factors of the FDCE into the SD domain. It is possible to prove that there exists an inverse function *ameanf-1* to the function *ameanf.*

From the *ameanf-1* function we can construct the *inverse semantics function* called *ameanz* which maps elements of the SD domain onto LO (see fig. 3). The function *ameanz* has the following property:

a = *amean* (wx) ==> = *ameanz* (a),

where wx $\equiv$ wy ($\equiv$ is a symbol of the equivalence relation), a ϵ SD,
wx, wy ϵ LO.

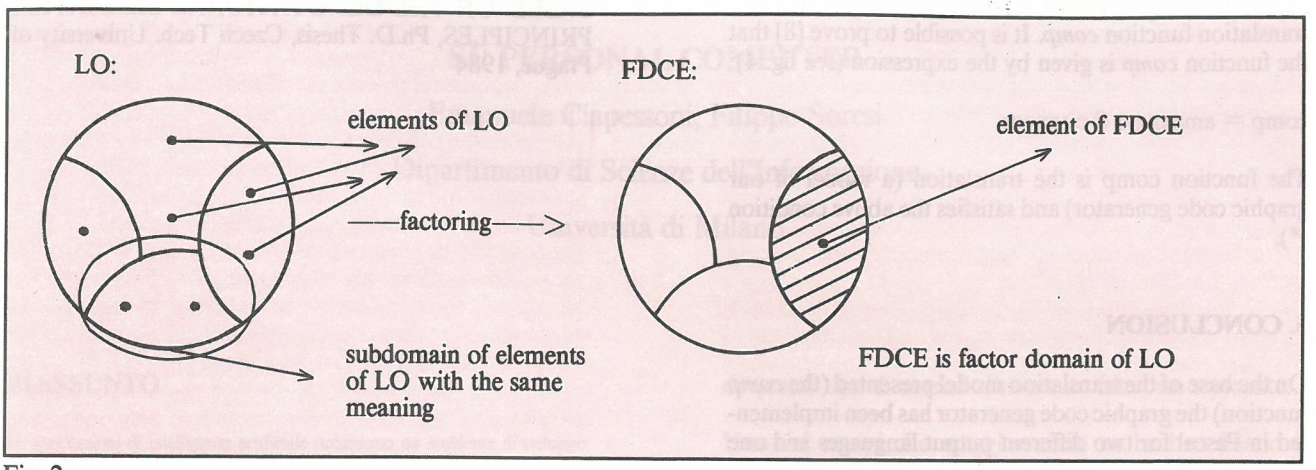


Fig. 2

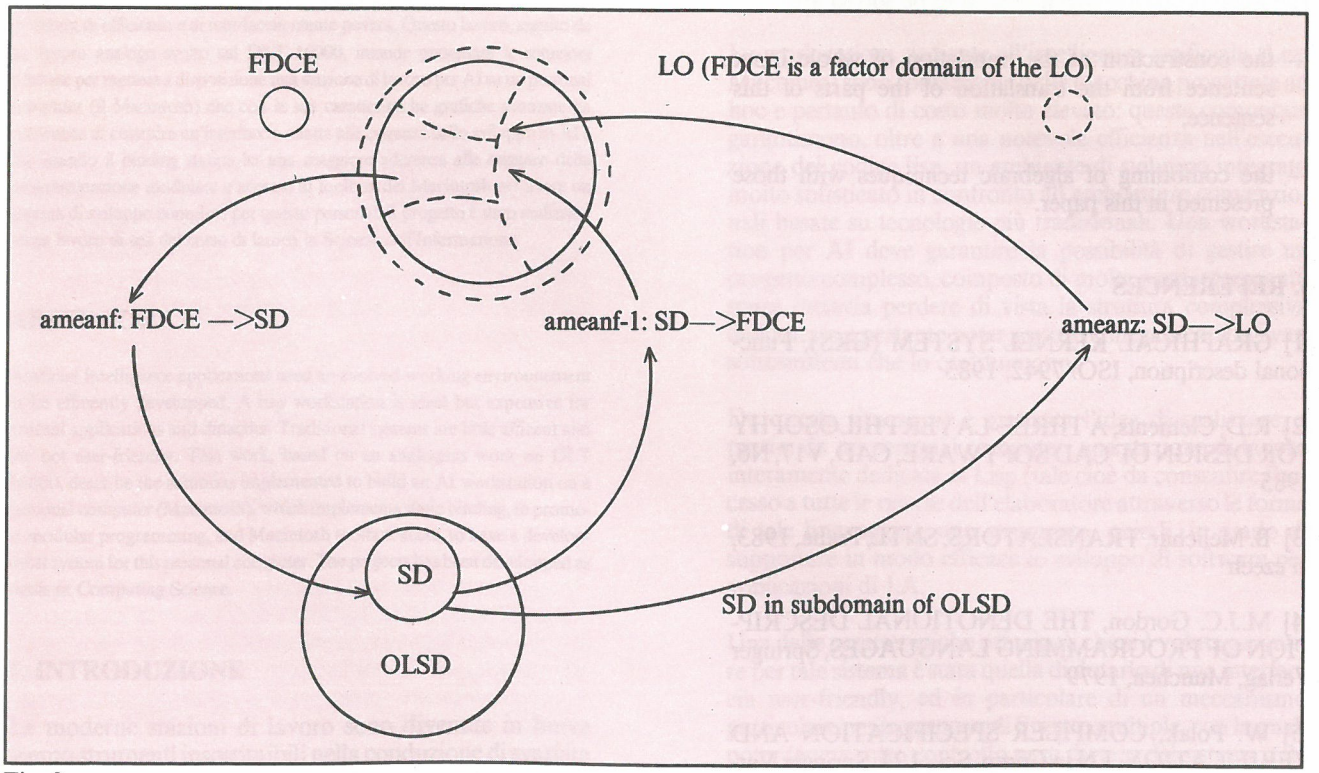


Fig. 3

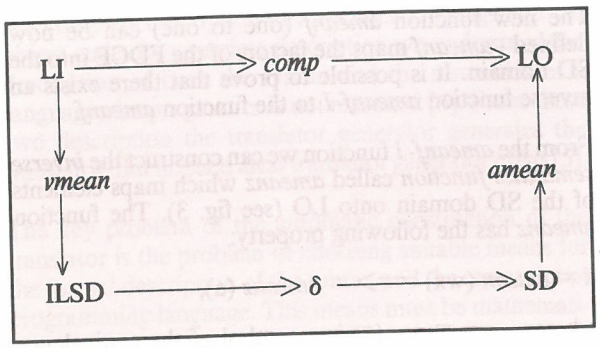


Fig. 4

From the functions *vmean, δ, amenz* we can construct the translation function *comp*. It is possible to prove [8] that the function *comp* is given by the expression (see fig. 4):

comp = ameanz o δ o vmean.

The function comp is the translation (a model of our graphic code generator) and satisfies the above condition (*).

5. CONCLUSION

On the base of the translation model presented (the *comp* function) the graphic code generator has been implemented in Pascal for two different putput languages and one input language.

The main problems which remain to be solved are:

— the construction of the translation of whole input sentence from the translation of the parts of this sentence,

— the combining of algebraic techniques with those presented in this paper.

6. REFERENCES

[1] GRAPHICAL KERNEL SYSTEM (GKS), Functional description, ISO/7942, 1985

[2] R.D. Clements, A THREE-LAYER PHILOSOPHY FOR DESIGN OF CAD SOFTWARE, CAD, V17, N6, 1985

[3] B. Melichar, TRANSLATORS, SNTL, Praha, 1983, in czech

[4] M.J.C. Gordon, THE DENOTIONAL DESCRIPTION OF PROGRAMMING LANGUAGES, Springer Verlag, Munchen, 1979

[5] W. Polak, COMPILER SPECIFICATION AND VERIFICATION, LN in Comp. Sci., 124, Springer Verlag, 1981

[6] B. Melichar, K. Mueller, THE AUTOMATIC TRANSLATOR WRITING, SOFSEM '78, VVS Bratislava, 1978, in czech

[7] R. Milne, Ch. Strachey, A THEORY OF PROGRAMMING LANGUAGES SEMANTICS, Chapman and Hall, London, 1976

[8] H. Ganzinger, TRANSFORMING DENOTATIONAL SEMANTICS INTO PRACTICAL ATTRIBUTE GRAMMARS, LN in Comp. Sci., 94, Springer Verlag, 1980

[9] I. Jelinek, THE GRAPHIC CODE GENERATOR BASED ON THE DENOTATIONAL SEMANTICS PRINCIPLES, Ph.D. Thesis, Czech Tech. University of Prague, 1984

ωLISP: UN AMBIENTE DI SVILUPPO PER A.I. SU PERSONAL COMPUTER

Emanuele Ciapessoni, Filippo Soresi
Dipartimento di Scienze dell'Informazione
Università di Milano

RIASSUNTO

Le applicazioni di intelligenza artificiale richiedono un ambiente di sviluppo evoluto per poter essere svolte in modo efficiente. Una stazione di lavoro dedicata unicamente al lisp (Lisp Machine) daltronde è troppo costosa per le normali applicazioni e per la didattica. Nelle architetture convenzionali si hanno problemi di efficienza e di interfaccia utente povera. Questo lavoro, seguito da un lavoro analogo svolto sul DLT 16000, intende presentare le soluzioni adottate per mettere a disposizione una stazione di lavoro per AI su un personal computer (il Macintosh) che con le sue caratteristiche grafiche avanzate ha consentito di costruire un'interfaccia adatta alle esigenze dello sviluppo in AI e che usando il binding statico ha una maggiore aderenza alle esigenze della programmazione modulare e accesso al toolbox del Macintosh per avere un sistema di sviluppo completo per questo personal. Il progetto è stato realizzato come lavoro di tesi del corso di laurea in Scienze dell'Informazione.

ABSTRACT

Artificial Intelligence applications need an evolved working environnement to be efficently developped. A lisp workstation is ideal but expensive for normal applications and didactics. Traditional systems are little efficent and are not user-friendly. This work, based on an analogous work on DLT 16000, describe the solutions implemented to build an AI workstation on a personal computer (Macintosh), which implements static binding, to promote modular programming, and Macintosh toolbox acces, to have a development system for this personal computer. The projects has been developped as thesis in Computing Science.

1. INTRODUZIONE

Le moderne stazioni di lavoro sono divenute in breve tempo strumenti insostituibili nella conduzione di svariate attività e per questo motivo la realizzazione e lo sviluppo di questo tipo di sistemi riscuote un interesse crescente.

Uno dei campi che maggiormente ne ha beneficiato è senza dubbio quello della Intelligenza Artificiale i cui risultati cominciano ad acquistare rilevanza in settori che esulano dall'ambito della ricerca pura.

Le workstations dedicate all'intelligenza artificiale (Lisp Machines) sono solitamente delle macchine progettate ad hoc e pertanto di costo molto elevato: queste comunque garantiscono, oltre a una notevole efficienza nell'esecuzione del codice lisp, un ambiente di sviluppo integrato molto sofisticato in confronto ad architetture convenzionali basate su tecnologie più tradizionali. Una workstation per AI deve garantire la possibilità di gestire un progetto complesso, composto di molte parti interagenti, senza tuttavia perdere di vista la struttura complessiva dello stesso e pertanto poter gestire i collegamenti tra i vari sottosinsiemi che lo costituiscono.

Da questa situazione è scaturita l'idea di realizzare, a partire da un personal computer, una stazione di lavoro interamente dedicata al Lisp (tale cioè da consentire l'accesso a tutte le risorse dell'elaboratore attraverso le forme di tale linguaggio), uno strumento, quindi, in grado di supportare in modo efficace lo sviluppo di software per applicazioni di I.A.

Una delle caratteristiche principali che si è voluta garantire per tale sistema è stata quella di dotarlo di una interfaccia user-friendly, ed in particolare di un meccanismo particolare per la gestione di finestre multiple, con le quali poter tenere sotto controllo parti diverse del sistema; tale requisito ha condizionato la scelta dell'elaboratore su cui

sviluppare il progetto, e l'Apple Macintosh è apparso il più idoneo a tale scopo, grazie alla presenza del mouse ed al firmware contenuto in ROM (toolbox).

Inoltre si è voluto rendere quanto più completo possibile l'accesso alle strutture dati dell'interprete, facendo sì che tutti gli oggetti manipolati fossero cittadini di prima classe, cioè oggetti lisp completamente manipolabili dall'utente. Questo ha spinto verso l'utilizzo di un regime di binding statico delle variabili che consente un maggior grado di modularità e una corretta gestione delle funzioni di ordine superiore, caratteristica essenziale dei moderni linguaggi funzionali. Inoltre gli ambienti di valutazione sono stati posti a livello degli oggetti lisp, in modo da poter aggirare il binding statico qualora ciò risultasse necessario.

All'utente finale questo sistema, battezzato "ωLisp", si presenta come un sistema multitasking, in cui cioè più processi possono essere presenti contemporaneamente.

L'utente interagisce con tali processi (ai quali è demandata ogni attività all'interno del sistema) attraverso le già citate finestre, ed entrambi appaiono all'utente come oggetti Lisp a tutti gli effetti.

Questo sistema rappresenta la naturale evoluzione di "wLisp" realizzato da Giuseppe Colli e Stefano Masselodi all'Istituto di Cibernetica (le cui caratteristiche sono esposte in [6]) e ne mantiene la struttura di base fondata su un analogo concetto di processo, nonché diversi aspetti implementativi, quale l'adozione di un garbage collector di tipo incrementale su una memoria organizzata a pagine (che, oltre ad ottimizzare l'uso di quest'ultima, risulta meno oneroso in termini computazionali), l'uso dei puntatori tipati e dei metodi di valutazione (che permette, da una parte la semplificazione dell'algoritmo di valutazione delle espressioni Lisp, e dall'altra garantisce una facile espandibilità del sistema), nonché la realizzazione di un analogo sistema per il trattamento degli errori e delle condizioni eccezionali, aspetti ampiamente trattati nel già citato articolo.

Il progetto è stato sviluppato come tesi di laurea del corso di Scienze di Informazione [3] e [11] svolte in collaborazione con il dott. E. Ciapessoni del Dipartimento di Informatica e il dott. L. Spampinato della Quinary. Le due tesi citate riguardano, una l'interfaccia utente del sistema, l'altra la struttura dell'interprete.

2. IL BINDING STATICO

Come già detto ciò che differenzia fondamentalmente ωLisp dal suo predecessore è l'adozione di un legame di tipo lessicale (binding statico) tra le variabili ed i rispettivi valori; questa scelta, fatta per allineare il sistema alle più moderne realizzazioni del settore (Scheme, Common Lisp che sono stati una notevole fonte di ispirazione durante il lavoro di specifica) e per poter utilizzare a pieno titolo le funzioni come cittadini di prima classe (con la possibilità di utilizzare funzioni di ordine superiore: funzionali, combinatori, ecc.), ha influenzato in varia misura lo sviluppo di ogni parte del progetto ed ha comportato la creazione di nuove e particolari strutture dati, nonché fondamentali modifiche all'algoritmo di valutazione delle espressioni Lisp.

È noto che un regime di binding statico facilita lo sviluppo di software, poiché è possibile programmare tenendo conto delle specifiche funzionali, ossia delle relazioni tra i valori di input e di output di un dato frammento di codice (a differenza di ciò che avviene nel caso del binding dinamico in cui occorre sempre tener conto di come una funzione è stata implementata prima di poterla utilizzare in un programma); ciò consente l'adozione di tecniche di programmazione modulare (che permettono la realizzazione di grossi programmi e di codice riutilizzabile), inoltre la fase di debugging risulta meno impegnativa poiché il comportamento del codice diventa più prevedibile, ed anche le eventuali modifiche successive, in fase di manutenzione, non comportano il rischio di imprevedibili side-effects presenti invece in ambito dinamico.

A fronte di tali vantaggi esiste però una maggiore complessità nella implementazione di tale sistema dovuta alla particolare gestione degli "ambienti di valutazione": questi oggetti, che contengono le associazioni nome-valore necessarie alla determinazione del valore delle variabili, ricoprono un ruolo fondamentale nel processo di valutazione delle espressioni Lisp, poiché queste ultime assumono significato solo in relazione ad un dato ambiente.

Schematizzando, possiamo raffigurare un ambiente come un oggetto formato da una tabella (frame) e da un puntatore (link statico) ad un altro ambiente: la ricerca del valore di una variabile viene fatta a partire da una dato frame seguendo questa catena di puntatori, fino a trovare un legame nome-valore corretto.

In generale questo è ciò che accade al momento di determinare il valore di una variabile libera: quando una funzione viene eseguita, viene creato un frame in cui i parametri formali vengono associati ai valori attuali degli argomenti; tale frame tuttavia non contiene alcuna relazione per le altre variabili che compaiono nel corpo della funzione, ed i loro valori andranno ricercati nei frames degli ambienti a cui esso è legato.

In una situazione di binding dinamico il puntatore di un ambiente fa sempre riferimento all'ambiente che lo precede nell'ordine di creazione, mentre per il binding statico (come in questo caso) occorre far riferimento all'ambiente in cui la funzione, la cui esecuzione ha generato il nuovo frame, è stata creata.

Per questo motivo, la valutazione di una Lambda espressione genera un oggetto chiamato "funzione" che associa al corpo dell'espressione stessa l'ambiente in cui avviene tale valutazione. Quando poi la funzione verrà applicata, il nuovo ambiente creato punterà all'ambiente associato alla funzione stessa.

L'ultimo ambiente generato in ordine di tempo riveste il

ruolo di "ambiente corrente" ed è quello da cui iniziare per la ricerca dei valori delle variabili.

Una operazione molto importante sugli ambienti, o più precisamente sull'ambiente corrente, è quello dell'aggiunta di una nuova relazione nome-valore: tale effetto si ottiene utilizzando la forma speciale

(DEFINE <var> <exp>)

che lega a <var> il valore dell'espressione <exp>.

Di questa forma esiste inoltre una sintassi particolare per la definizione di funzioni che mostra direttamente la struttura della funzione quando questa verrà applicata ai suoi argomenti; l'espressione

(DEFINE (<fun> <vars>) <bodies>)

viene espansa in

(DEFINE <fun> (lambda <vars> <bodies>))

dove <fun> può a sua volta essere una lista per la definizione della funzione e così via; per cui ad esempio

(DEFINE ((comp f g) x)(f (g x)))

subisce una doppia espansione in:

(DEFINE (comp f g) (lambda(x) (f (g x))))

e in:

```
(DEFINE comp (lambda (f g)
        (lambda (x) (f (g x)))))
```

che è la classica definizione del combinatore composizione di due funzioni. Si noti che in ωLisp si ha in corretto trattamento di questi combinatori grazie all'uso del binding statico, che consente l'uso di funzioni di ordine superiore.

3. GLI AMBIENTI COME CITTADINI DI PRIMA CLASSE

Da ciò che si è detto precedentemente risulta evidente l'importanza del ruolo svolto dagli ambienti nel processo di valutazione di espressioni Lisp. Tali oggetti tuttavia sono in genere nascosti all'utente che vede soltanto l'effetto finale della loro gestione.

Si è quindi pensato di fornire un ulteriore grado di libertà nell'uso del sistema dando la possibilità di accedere e manipolare tramite alcune funzioni primitive tali oggetti come se fossero strutture dati Lisp, in modo da promuoverli a "cittadini di prima classe' al pari degli atomi, dei numeri, delle funzioni, ecc...

Per l'implementazione di tale meccanismo si è fatto ricorso alla strategia dei puntatori tipati esposta in [6]: una volta definito un identificatore per il tipo "ambiente" quest'ultimo può essere assunto come valore da parte di una variabile.

Le possibilità di tale sistema sono molto interessanti: diventa ad esempio possibile passare un ambiente come argomento ad una funzione, e modificare completamente la normale semantica di un programma con delle opportune manipolazioni sugli ambienti.

L'utente ha a disposizione tre variabili globali:

a) Global_env: ha come valore l'ambiente globale,

b) Base_env: che ha come valore l'ambiente base del processo corrente (la cui utilità verrà chiarita in seguito),

c) Current_env che assume come valore l'ambiente corrente; per operare su tali variabili e su altre dello stesso tipo generate dall'utente si possono utilizzare delle opportune funzioni primitive che vengono ora esposte:

EVAL
Questa funzione primitiva del Lisp standard è stata modificata nella sintassi e nella semantica per poter operare con gli ambienti; la sua forma estesa si presenta così:

(EVAL <exp> [env])

Il cui risultato è quello della valutazione di <exp> nell'ambiente [env], che viene trattato come parametro opzionale; se questo secondo argomento non è specificato viene preso per default l'ambiente base del processo che valuta tale espressione.

MAKE_ENVIRON
È una forma speciale del linguaggio:

(MAKE_ENVIRONMENT <exp1> ... <expN>)

Exp1,...,expN sono espressioni che vengono valutate. Tutte le variabili che vengono definite (tramite ad esempio la forma speciale DEFINE) nella valutazione di tali espressioni vanno a far parte del nuovo ambiente generato. Tale ambiente è staticamente legato all'ambiente in cui Make_environment è stata chiamata.

Come esempio supponiamo di valutare questa espressione nell'ambiente globale:

```
(define alfa (make_environment
        (define x 5)
        (define y 10)
        (define z "abc")))
```

Il risultato sarà un legame nell'ambiente globale tra Alfa ed un oggetto di tipo ambiente avente come frame {X <—> 5,y <—> 10,Z <—> "abc""}.

ACCESS
È una forma speciale del linguaggio:

(ACCESS <sym> <env>)

Prende come parametri un simbolo ed un ambiente: ritorna il valore del simbolo nell'ambiente specificato.

Esempio:

(access x alfa) → 5
(access alfa alfa) → ★ENVIRONMENT★

Come si può capire dall'esempio se la ricerca fallisce nell'ambiente stesso essa prosegue negli ambienti che lo racchiudono staticamente: in questo caso "alfa" ha un legame in Global-Env ed il valore di "alfa" è un ambiente.

FRAME_BINDINGS
È una funzione primitiva:

(FRAME_BINDINGS <env>)

Ritorna in forma di lista Lisp i legami nome-valore contenuti in env.

Esempio:

(frame_bindings alfa) → ((x 5)(y 10)(z "abc"))

FRAME_PROCEDURE
È una funzione primitiva:

(FRAME_PROCEDURE env)

Ritorna la funzione che è stata applicata per produrre l'ambiente Env.

FRAME_PARENT
È una funzione primitiva

(FRAME_PARENT env)

Ritorna l'ambiente in cui Env è stato creato.

Queste funzioni rappresentano per l'utente uno strumento molto versatile: attraverso di esse, ad esempio, è possibile utilizzare un approccio data-driven alla costruzione dei programmi, tecnica che consiste nel costruire un ambiente che contenga le definizioni delle operazioni che corrispondono a un certo "tipo": concretamente è possibile, utilizzando la funzione MAKE_ENVIRONMENT costruire il tipo della "frazioni" in cui vengono definite alcune operazioni, ad esempio la somma e la moltiplicazione di frazioni:

```
(define frazioni(MAKE_ENVIRONMENT
        (define(plus x y)(.....))
        (define(mult x y)(.....))
        .....))
```

e con la stessa tecnica possiamo creare l'ambiente, e quindi il tipo, degli "interi", dei "complessi", etc.

In questa situazione, data una funzione (type <arg>) che ritorna il tipo e quindi l'ambiente che identifica l'insieme di appartenenza di <arg>, ed una funzione (contents <arg>) che ritorna il valore di <arg>, è possibile definire una funzione globale "plus":

```
(define(plus x y)((ACCESS plus (type x))
        (contents x)
        (contents y))
```

la cui semantica dipende dall'insieme di appartenenza degli argomenti: in questo modo il nome di un ambiente ricopre la funzione di identificatore di tipo.

Utilizzando lo stesso meccanismo è possibile inoltre creare dei "packages" [1], sfruttando gli ambienti per inglobare informazioni che devono rimanere locali; tali informazioni possono essere successivamente reperite utilizzando la possibilità della funzione EVAL o ACCESS di accettare un ambiente come argomento. In questo modo si può ottenere un buon livello di modularizzazione e incapsulazione del programma con i noti vantaggi che questo approccio comporta.

4. I PROCESSI

Esistono particolari situazioni in cui è preferibile (se non necessario) scomporre una data attività in azioni che possono essere svolte parallelamente ed in modo autonomo.

Tali "azioni" hanno un corrispettivo nel concetto di processo adottato da ωLisp (ereditato da wLisp): una entità attiva la cui evoluzione è rappresentata dalla successione di passi di valutazione di una espressione Lisp (azione) ad esso associata.

L'instanziazione di un processo avviene generando un contesto indipendente in cui l'espressione viene valutata. La fine della valutazione implica la distruzione di tale contesto e la morte del processo. Ogni attività all'interno del sistema è svolta da una di tali entità.

Al momento dell'inizializzazione esiste nel sistema un unico processo; a cui è associata una funzione che esegue ripetutamente un ciclo di lettura-valutazione-scrittura attraverso la finestra di default (anch'essa creata al momento dell'inizializzazione). In questa configurazione il comportamento del sistema è quello di un semplice interprete Lisp.

La creazione di altri processi avviene tramite la funzione "PROCESS"; poiché un processo è un oggetto Lisp, può essere attribuito come valore ad una variabile che diventa il "nome" del processo.

In una situazione in cui coesistono più processi, questi vengono attivati secondo uno schema time-sharing, tuttavia viene data all'utente la possibilità di alterare questo

stato di cose utilizzando particolari funzioni (esposte in [6]) che ne regolano esplicitamente la sincronizzazione.

Perchè i processi possano essere utilizzati allo scopo di organizzare meglio una data attività è necessario che questi creino un ambiente autonomo in cui valutare l'espressione ad essi associata, pur mantenendo la possibilità di stabilire tra di loro un livello di comunicazione.

Per questo motivo si sono dovute stabilire determinate convenzioni per regolare il rapporto tra i processi stessi ed i già citati ambienti di valutazione.

Al momento della generazione di un processo, a questo viene associato un nuovo ambiente (inizialmente vuoto), che ne rappresenta l'ambiente base, creando una "intercapedine" tra il processo ed il resto del sistema: ciò garantisce la correttezza delle valutazioni successive (che sembreranno effettuate nell'ambiente in cui il processo è stato creato) e permette la privatezza delle definizioni di nuove variabili i cui legami verranno posti nell'ambiente base.

Per quanto riguarda lo schema di comunicazione, ne esiste uno automatico tra un processo "padre" (esecutore di una "Process") ed un processo figlio (il processo generato): quest'ultimo eredita, in parte o totalmente, l'ambiente del padre; in tal modo il processo padre e il processo figlio riescono a comunicare tramite l'accesso alle variabili del processo padre che sono visibili dal processo figlio; con questa tecnica è concepibile una schema in cui in processo genera due o più processi figlio e si comporta poi come gestore dello scambio dati tra i vari processi figli; tale meccanismo è tuttavia limitato, poiché consente solo l'accesso e la modifica di valori di variabili del processo padre e non l'uso di quella variabili che il processo figlio ha definito nel proprio ambiente.

Un sistema più efficace può essere però attuato sfruttando le caratteristiche della funzione EVAL viste in precedenza e le funzioni di manipolazione degli ambienti: supponiamo che esistano due processi e che il primo esegua

(eval (define y 10) GLOBAL_ENV)

che provoca l'aggiunta del legame y ↔ 10 all'ambiente globale; per accedere a tale valore di 'y' il secondo processo non dovrà fare altro che valutare l'espressione

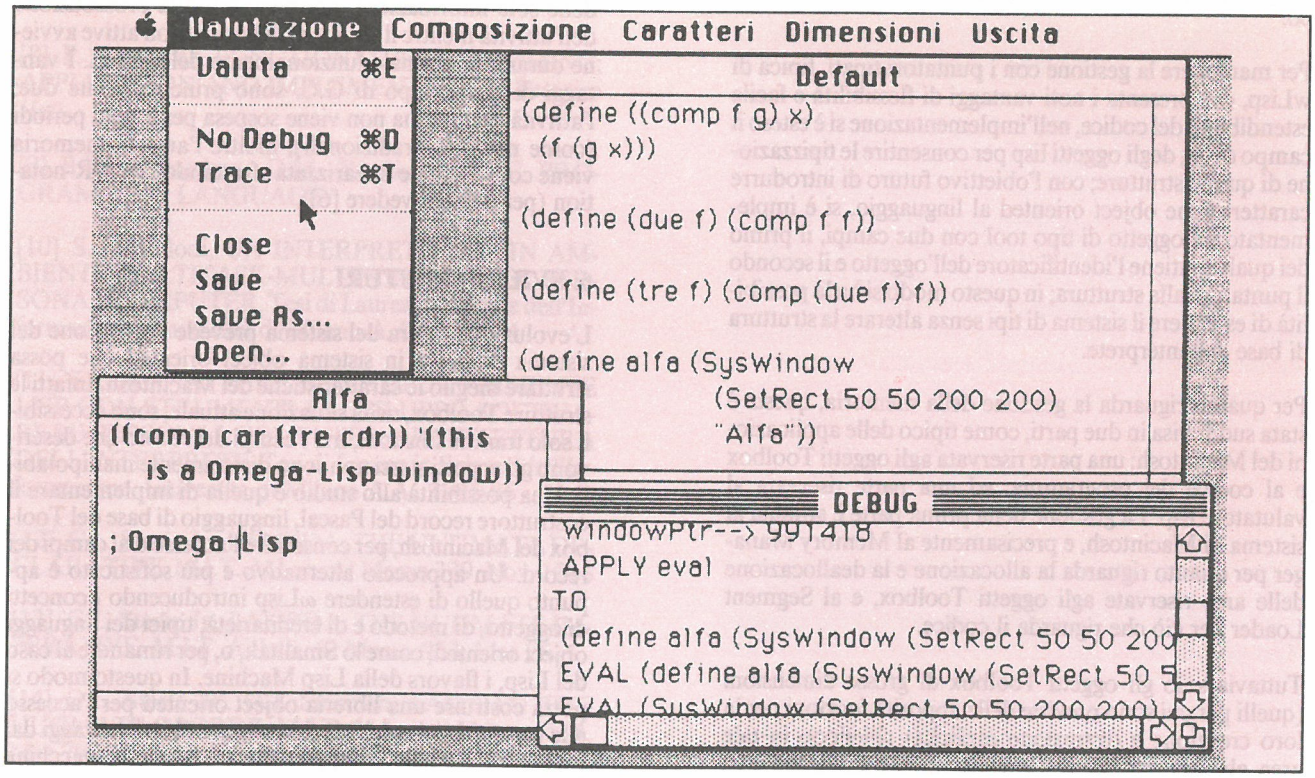
(access y GLOBAL_ENV)

Con questa tecnica è possibile effettuare delle manipolazioni degli ambienti molto sofisticate; naturalmente dovrà essere l'utente, con una corretta programmazione, ad evitare eventuali problemi di inconsistenza dei dati.

5. L'INTERFACCIA UTENTE

L'interfaccia utente realizzata per ωLisp è stata concepita secondo una filosofia, ormai affermata, che impone la creazione di un ambiente di lavoro che, oltre a garantire un accesso rapido alle risorse dell'elaborazione risulti gradevole e di semplice utilizzo.

Lo standard a cui ci si è uniformati è quello tipico delle applicazioni in ambiente Macintosh, che comporta l'uso del mouse, la presenza dei menu, la suddivisione dello schermo in più finestre sovrapponibili.



All'inizio di una sessione di lavoro all'utente si presenta questa situazione: sullo schermo compare un'unica finestra (di default) su cui è possibile editare un testo attraverso un editor mouse-driven, ed un insieme di menu le cui funzioni principali sono quelle relative al richiamo del valutatore Lisp su espressioni precedentemente selezionate, compiere operazioni di IO con la memoria di massa, impartire comandi all'editor, determinare fonte e dimensioni dei caratteri del testo editato; l'utente può quindi scrivere liberamente qualsiasi tipo di espressione e decidere in seguito cosa valutare, richiamando esplicitamente l'interprete.

Ciò è dovuto alla particolare funzione, associata al processo iniziale, che esegue ripetutamente un ciclo di lettura-valutazione-scrittura nella finestra di default utilizzando, al posto della usuale "read" che riconosce la terminazione dell'espressione corrente, la funzione "sysread", che dopo aver ricevuto dall'utente un comando eplicito di valutazione inizia a leggere l'espressione selezionata con il mouse. Se tuttavia tale situazione non fosse gradita all'utente questi, ad esempio, può creare una nuova finestra, creare un nuovo processo, associandovi una funzione che utilizzi la classica "read" attraverso la finestra appena creata (sia la funzione "read" che la "sysread" accettano come parametro una finestra) per ottenere un ambiente Lisp di tipo tradizionale.

In ωLisp tutti gli oggetti grafici del Macintosh sono visti come oggetti Lisp: le finestre che possono essere create e associate a uno o più processi per quanto riguarda l'I/O di questi, possono essere dei vari tipi previsti dal Toolbox, e si ha la possibilità di utilizzare le primitive grafiche del toolbox per poter utilizzare ωLisp come sistema di sviluppo.

Per mantenere la gestione con i puntatori tipati, tipica di wLisp, che presenta i noti vantaggi di flessibilità e facile estendibilità del codice, nell'implementazione si è esteso il campo di tag degli oggetti lisp per consentire le tipizzazione di queste strutture; con l'obiettivo futuro di introdurre caratteristiche object oriented al linguaggio, si è implementato un oggetto di tipo tool con due campi, il primo dei quali contiene l'identificatore dell'oggetto e il secondo il puntatore alla struttura; in questo modo si ha la possibilità di estendere il sistema di tipi senza alterare la struttura di base dell'interprete.

Per quanto riguarda la gestione della memoria, questa è stata sudduvisa in due parti, come tipico delle applicazioni del Macintosh: una parte riservata agli oggetti Toolbox e al codice del programma, ed una parte riservata al valutatore lisp. La gestione della prima parte è affidata al sistema di Macintosh, e precisamente al Memory Manager per quanto riguarda la allocazione e la deallocazione delle aree riservate agli oggetti Toolbox, e al Segment Loader per ciò che riguarda il codice.

Tuttavia solo gli oggetti Toolbox di grosse dimensioni (quelli per cui sono previste delle apposite funzioni per la loro creazione e distruzione esplicita) risiedono in tale area, gli oggetti più piccoli vengono invece tenuti nell'area riservata alle strutture lisp vere e proprie, ed è garbage collector che si preoccupa di riutilizzare l'area da essi occupata.

L'organizzazione della memoria dipende dalla versione di Macintosh utilizzata: nella versione a 512K i primi 50K sono riservati allo heap di sistema; lo spazio rimanente viene così utilizzato: 70K riservati al codice, 50K riservati agli oggetti Toolbox, 255K riservati al sistema lisp, mentre lo spazio rimanente viene utilizzato dallo stack di sistema, dalla memoria riservata al video, ecc., mentre nella versione "plus" l'area riservata agli oggetti Toolbox sale a 80K e quella riservata al lisp a 732K.

L'area lisp è stata organizzata a pagine da 512 bytes (500 pagine su Macintosh a 512K, 1440 sulla versione "plus") per motivi di efficienza e per una futura gestione della memoria virtuale. Ogni pagina contiene 128 celle Lisp da 32 bits di cui 24 riservati ad un indirizzo mentre gli 8 bits più significativi sono riservati ad informazioni riguardanti gli oggetti puntati (puntatori tipati); in particolare i due bits più significativi vengono utilizzati dalla CDR-notation: una tecnica che tramite la codifica del puntatore al CDR di una lista consente un risparmio che può raggiungere il 50% rispetto alla memorizzazione tradizionale delle liste. Su una memoria così strutturata agisce un garbage collector di tipo incrementale (adattamento dell'algoritmo di Baker [3]): l'insieme delle pagine viene diviso logicamente in due gruppi in modo da creare due "semispazi" distinti; l'allocazione di nuova memoria viene effettuata su uno solo di questi semispazi, quando questo è completo si inizia ad utilizzare il secondo; in questo momento l'attività del sistema viene sospesa per un breve periodo per permettere la copiatura sul nuovo semispazio delle sole informazioni indispensabili alla prosecuzione dell'attività mentre il recupero delle aree non attive avviene durante il normale funzionamento del sistema. I vantaggi di questo tipo di G.C. sono principalmente due: l'attività del sistema non viene sospesa per lunghi periodi (come nei G.C. tradizionali), inoltre l'area di memoria viene compattata e linearizzata sfruttando la CDR-notation (per dettagli vedere [6]).

6. SVILUPPI FUTURI

L'evoluzione futura del sistema prevede l'estensione del sistema di tag a in sistema object oriented che possa sfruttare meglio le caratteristiche del Macintosh. Infatti le strutture Toolbox, nella situazione attuale, sono accessibili solo tramite funzioni, ma i campi dei record che descrivono gli oggetti stessi non sono direttamente manipolabili. Una possibilità allo studio è quella di implementare il costruttore record del Pascal, linguaggio di base del Toolbox del Macintosh, per consentire l'accesso a ai campi dei record. Un approccio alternativo e più sofisticato è appunto quello di estendere ωLisp introducendo i concetti di oggetto, di metodo e di ereditarietà, tipici dei linguaggi object oriented, come lo Smalltak, o, per rimanere al caso del Lisp, i flavors della Lisp Machine. In questo modo si potrà costruire una libreria object oriented per l'accesso alle caratteristiche del Macintosh, con chiari vantaggi dal punto di vista dello sviluppo software su questa macchina

e con la possibilità di utilizzare a pieno titolo questo potente paradigma computazionale nello sviluppo di applicazioni di A.I.

7. BIBLIOGRAFIA

[1] H. Abelson, G.J. Sussman, STRUCTURE AND IMPLEMENTATION OF COMPUTER PROGRAMS, MIT Press, 1986

[2] J. Allen, ANATOMY OF LISP, McGraw-Hill, 1978

[3] H.G. Baker, LIST PROCESSING IN REAL TIME ON A SERIAL COMPUTER, CACM 21-4

[4] M.L. Caprera, INTEGRAZIONE DI UN AMBIENTE LISP CON STRUMENTI DI SVILUPPO SOFTWARE SU PERSONAL COMPUTER: INTERAZIONE UTENTE SISTEMA, Tesi di Laurea in Scienze dell'Informazione, Università di Milano A.A., 1983-84

[5] G. Colli, UN SISTEMA OPERATIVO LISP MULTITASK E MULTIWINDOW PER PERSONAL COMPUTER, Tesi di Laurea in Scienze dell'Informazione, Università di Milano A.A., 1983-84

[6] G. Colli, L. Spampinato, UN SISTEMA OPERATIVO ED UN INTERPRETE LISP MULTITASK E MULTIWINDOW PER PERSONAL COMPUTER, Note Di Software N° 29/30, Dicembre 1985

[7] A.N. Haberman, INTRODUCTIO TO O.S. Design, Sra, 1976

[8] P. Henderson, FUNCTIONAL PROGRAMMING; APPLICATION AND IMPLEMENTATION, Prentice Hall

[9] B.W. Kernighan, D.M. Ritchie, THE C PROGRAMMING LANGUAGE

[10] S. Messelodi, UN INTERPRETE LISP IN AMBIENTE MULTITASK-MULTIWINDOW PER PERSONAL COMPUTER, Tesi di Laurea in Scienze dell'Informazione, Università di Milano A.A., 1983-84

[11] F. Soresi, INTEGRAZIONE DI UN AMBIENTE LISP CON STRUMENTI DI SVILUPPO SOFTWARE SU PERSONAL COMPUTER: ARCHITETTURA DELL'INTERPRETE, Tesi di Laurea in Scienze dell'Informazione, Università di Milano A.A., 1985-86

[12] G.L. Steele Jr., LAMBDA: THE ULTIMATE DECLARATIVE, M.I.T. Al Lab, Al Memo 379, Nov. 1976

[13] G.L. Steele Jr., LAMBDA: THE ULTIMATE IMPERATIVE, M.I.T. Al Lab, Al Memo 953, Mar 76

[14] G.L. Steele Jr., G.J. Sussman, LAMBDA: THE ULTIMATE OPCODE, M.I.T. Al Lab, Al Memo 514, Mar 79

[15] G.L. Steele Jr., G.J. Sussman, SCHEME: A DIALECT OF LISP, M.I.T. Al Lab, Al Memo 452, Gen 78

[16] G.J. Sussman, LISP, PROGRAMMING AND IMPLEMENTATION, in FUNCTIONAL PROGRAMMING. AN ADVANCED COURSE, Cambridge Press 1982

AVVENIMENTI

REWRITING TECHNIQUES AND APPLICATIONS

Maria Paola Bonacina

Dipartimento di Scienze dell'Informazione

Università di Milano

Dal 25 al 27 maggio 1987 l'Università di Bordeaux 1 ha ospitato la Seconda Conferenza Internazionale su Rewriting Techniques and Applications (RTA '87). La decisione di organizzare una conferenza biennale sulle tecniche di riscrittura è storia recente: la prima edizione si tenne solo due anni fa, nel 1985, a Digione.

Una conferenza di questo tipo sanziona la presenza di un'area di ricerca autonoma e ben definita, sebbene profondamente collegata ad altri indirizzi di ricerca in Computer Science con cui condivide radici storiche o metodi o problemi. Le radici storiche degli studi sui sistemi di riscrittura sono ben più lontane nel tempo di quanto le recenti date di questi avvenimenti possano far pensare. La storia dei sistemi di riscrittura ha molto in comune con quelle di aree con ricca tradizione come la programmazione funzionale, la teoria dei linguaggi formali e la teoria degli automi: la logica sottostante ai sistemi di riscrittura è la logica equazionale come per la programmazione funzionale; il meccanismo computazionale della riscrittura è lo stesso del λ—calcolo, degli algoritmi di Markov, delle grammatiche di linguaggi formali; le riduzioni tra termini corrispondono alle transizioni tra stati in un automa.

Tuttavia gli studi sui sistemi di riscrittura in quanto tali si sono progressivamente intensificati nella seconda metà degli anni Settanta e ancor più nella prima metà degli anni Ottanta fino al risultato delle conferenze di Digione e di Bordeaux. Questa crescita si può interpretare come parte del processo di evoluzione dei linguaggi di programmazione e più in generale dell'interfaccia uomo-macchina, che segue due direzioni fondamentali: da un lato la realizzazione di interfacce sempre più amichevoli, quotidiane, in una parola informali, dall'altro la progettazione di linguaggi sempre più vicini alla matematica e alla logica, quindi estremamente rigorosi e formali. I sistemi di riscrittura dovrebbero contribuire alla seconda direzione come linguaggio puramente equazionale. Questi due indirizzi coesistono grazie ad una organizzazione gerarchica in cui i sistemi di riscrittura potrebbero rappresentare un linguaggio di basso livello, una sorta di assembler, al di sopra del quale si costruiscono linguaggi via via più estesi e piacevoli da usare. Questa linea richiede la realizzazione in hardware, con architetture parallele di nuova concezione, delle operazioni di riscrittura.

La conferenza di Bordeaux ha contribuito a rafforzare la percezione di queste linee di sviluppo più grazie alle conversazioni fuori programma che agli interventi ufficiali, tutti dedicati a singoli problemi tecnici del settore, con l'eccezione delle due invited lectures di Michael O'Donnell e Jean Pierre Jouannaud: la prima ha riportato l'esperienza raccolta durante un lungo lavoro di implementazione di un linguaggio equazionale; la seconda ha mostrato la crescita della teoria dei sistemi di riscrittura in termini di eleganza concettuale e pulizia formale, analizzando l'evoluzione dei metodi di prova di correttezza applicati alla procedura di Knuth-Bendix, presentata come motore inferenziale sui sistemi di riscrittura.

Alla procedura di Knuth-Bendix erano dedicati altri cinque interventi (Ursula Martin, Leo Bachmair & Nachum Dershowitz, Richard Göbel, Helene Kirchner).

Un solo intervento (R. Ramesh & I.V. Ramakrishnan) è

stato dedicato al problema della concezione di algoritmi efficienti su architetture parallele per il pattern matching, operazione di base della riscrittura: anche un solo intervento di questo tipo è parso significativo dal momento che la maggior parte dei convenuti appartiene ad aree di ricerca piuttosto lontane da quella degli algoritmi paralleli per motivi di tradizione storica.

Il banchetto offerto ai partecipanti presso il castello di Vayres, vicino a Bordeaux, la sera del 26, è stato l'occasione per annunciare che il prossimo appuntamento per RTA si terrà tra due anni negli Stati Uniti, probabilmente in una località della East Coast, con chairman Nachum Dershowitz.

I proceedings della conferenza sono stati pubblicati a cura di Pierre Lescanne, chairman di RTA '87 con David Plaisted, nel volume 256 della serie Lecture Notes in Computer Science di Springer Verlag.

INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE

MILANO 23 - 28 AGOSTO 1987

Francesco Gardin (^)(*), Alessandro Malacart (*)

Si è svolto a Milano tra il 23 ed il 28 agosto il convegno International Joint Conference on Artificial Intelligence (IJCAI), che rappresenta l'appuntamento principale per i ricercatori e gli operatori di tutto il mondo impegnati sul fronte dell'Intelligenza Artificiale e che è ormai giunto alla sua decima edizione.

Con i suoi oltre 2700 partecipanti, ha rappresentato un'ulteriore conferma dell'enorme interesse che tale disciplina suscita sia a livello accademico che industriale. Tale partecipazione infatti, pur non essendo per evidenti motivi logistici all'altezza dell'edizione precedente (Los Angeles 1985), deve essere valutata alla luce del fatto che si tratta di un convegno strettamente scientifico, con temi avenzati difficilmente diegeribili da un pubblico non iniziato.

Proprio per consentire una maggiore partecipazione di pubblico, il programma prevedeva dodici tutorial su argomenti di carattere generale attinenti agli sviluppi dell'Intelligenza Artificiale (IA) che, a parte un paio dove la partecipazione è stata decisamente scarsa, hanno avuto un notevole successo, a riprova del fatto sia che questo tipo di attività ben si sposa con un congresso scientifico sia che il numero di persone che si avvicinano all'IA è in continuo aumento.

Sul piano puramente scientifico, sono stati ammessi al congresso circa 250 relatori rigorosamente suddivisi in otto aree di interesse. A questo proposito è interessante notare l'evoluzione, attraverso le varie edizioni del convegno, dei temi proposti, che evidenzia uno sviluppo di linee di ricerca sempre meglio definite. Rispetto, ad esempio, all'ultima edizione il dato più eclatante è stata la scomparsa di un topic specifico sui Sistemi Esperti, decisione che ha favorito quelle esperienze applicative distintesi per soluzioni originali rispetto alla Rappresentazione della Conoscenza e all'Architettura e Linguaggi per sistemi di IA. Scomparsi anche temi 'storici' della ricerca nel settore, come ad esmpio la Programmazione Logica, la Dimostrazione di Teoremi, il Planning e la Programmazione Automatica, confluiti via via in temi più generali in cui hanno assunto un carattere più metodologico che di specifico interesse per la ricerca.

Un'osservazione abbastanza ricorrente tra gli addetti ai lavori è stata quella relativa all'estrema specificità dei lavori presentati, che sebbene da un punto di vista puramente tecnico erano nella maggior parte dei casi ineccepibili, non erano rivolti, in generale, a temi di sintesi. Per una disciplina alla ricerca di teorie unificanti questo atteggiamento estremamente specialistico forse non è esattamente quanto sarebbe richiesto oggi come sforzo intellettuale. È auspicabile che il comutato dell'IJCAI tenga presente considerazioni di questo tipo nella politica delle

(^) Dipartimento di Scienze dell'Informazione - Università di Milano
(*) Artificial Intelligence Software S.p.A. - Milano

decisioni relative agli argomenti da privilegiare.

In confortante aumento il numero di lavori italiani ammessi, e più in generale si può ben dire che si è registrato un lieve spostamento dell'asse di interesse in favore delle attività di ricerca europee. Ancora insufficiente invece, facendo riferimento al volume di investimenti in corso, la presenza giapponese.

L'apertura dei lavori era destinata quotidinamente alle 'invited lectures' che hanno consentito di ottenere un quadro abbastanza preciso sullo stato della ricerca ufficiale sulla tecnologia. La prima giornata è stata inaugurata da una relazione di Oliver Faugeras dedicata alla sperimentazione in corso presso il laboratorio dell'INRIA sui sistemi di visione artificiale in 3D. Il secondo giorno ha visto un invited talk di Gerald Gazdar relativo ai problemi di comprensione del linguaggio naturale, e in particolare sull'evoluzione dei formalismi nelle grammatiche di tipo dichiarativo. La terza giornata si è aperta invece con l'autorevole contributo di Douglas Lenat che ha esposto alcuni capisaldi della ricerca in corso presso l'MCC relativa ad un migliore comprensione del concetto di Conoscenza, ricerca che rappresenta forse la punta più avanzata nello studio di quel misterioso fenomeno che passa sotto il nome di 'common sense reasoning'.
Il penultimo lecturer è stato Hideo Aiso che ha relazionato circa lo stato dell'arte e le prospettive delle architetture hardware dedicate alla soluzione dei problemi di IA.
Il ciclo si è concluso con l'atteso intervento di Chandrasekaran, Ohio University, che ha presentato una proposta originale di approccio alla rappresentazione della conoscenza, in aperta polemica con gli attuali e consolidati formalismi (regole, frames, ecc.), tendendo quindi la mano a tutti coloro che da tempo richiedono strumenti formali che forniscono migliore ausilio alla delicata fase di Acquisizione delle Conoscenze. La proposta consiste in una 'Architettura Funzionale' per Sistemi Intelligenti con caratteristiche di elevata flessibilità, indispensabili, sostiene Chandrasekaran, per una onesta e proficua attività di Knowledge Ingeneering.

Sicuramente rilevante è stata anche la partecipazione di pubblico alla 'exhibition' organizzata in parallelo al congresso scientifico. Questa parte commerciale della manifestazione è stata caratterizzata da una folta presenza di espositori, superiore anche alla precedente edizione, che nel caso degli italiani ha rappresentato una piacevole sorpresa, sia per numero che per qualità dei prodotti presentati. Quetso sta ad indicare che il tanto decantato passaggio dall'accademia all'industria di questa tecnologia, che tante preoccupazioni aveva destato in Italia, data l'esiguità del numero di accademici di questo settore, e quindi del limitato numero di persone che potevano essere formate in un immediato futuro, in relatà è avvenuto, anche se in molti casi l'industria si è dovuta organizzare rapidamente formando il suo personale interno o stabilendo contatti con università straniere. È stato infatti veramente interessante notare il grado di professionalità raggiunto dalle aziende italiane presenti che, in alcuni casi, se dotate delle adeguate strutture commerciali potrebbero tranquillamente proporre i loro prodotti allo stesso mercato americano.

Come ad ogni IJCAI è stato premiato un ricercatore distintosi particolarmente per la propria attività scientifica. In questa edizione IJCAI-87 il ricercatore prescelto è stato Johan De Kleer, conosciuto per i suoi lavori sulla truth maintenance, la diagnostica, il calcolo delle confluenze e la fisica qualitativa. È appunto alla fisica qualitativa ed al ragionamento qualitativo in generale che lo 'speech' di De Kleer è stato dedicato. Si è trattato di una presentazione estremamente chiara dell'argomento, anche se in alcuni casi troppo divulgativa. La scelta da parte del comitato IJCAI di questo ricercatore, oltre indubbiamente ai suoi meriti scientifici in senso lato, è stata però il riconoscimento indiretto dell'importanza che sta assumendo all'interno dell'IA il settore del ragionamento qualitativo.

Durante la conferenza il comitato dell'IJCAI ha deciso di tenere l'IJCAI del 1991 in Australia, più precisamente a Melbourne, con conseguente immediata presentazione di dimissioni da parte del rappresentante francese che invece si vedeva rifiutare per la seconda volta Parigi come sede della manifestazione (la prima volta a Karlruhe nel 1983 la scelta era caduta su Milano).
Il prossimo IJCAI-89 si terrà invece negli Stati Uniti, a Detroit.

L'organizzazione, curata dal Prof. Somalvico, è stata ineccepibile ed ha fatto ben figurare l'Italia come paese in grado di ospitare manifestazioni internazionali di questo livello. L'unico neo è stato il diluvio del secondo giorno che ha causato l'allagamento di alcune sale, con conseguente annullamento di alcune presentazioni, che però sono state tempestivamente riprogrammate per il giorno successivo.

RECENSIONI

A GUIDE TO EXPERT SYSTEMS, Donald A. Waterman, Addison-Wesley Publishing Company, 1986

Da quando hanno cominciato ad emigrare dai laboratori di ricerca di Intelligenza Artificiale al campo applicativo i primi risultati concreti sulla tecnologia dei sistemi esperti, si è andato creando, com'era prevedibile, un insieme di aspettative e nel contempo di scetticismo sulle possibilità di utilizzo ed in genere sulle potenzialità di questi strumenti.

In ogni caso l'interesse è ormai talmente elevato da giustificare l'intervento di specialisti del settore nell'opera di informazione presso l'utenza non specializzata sullo stato dell'arte di questa tecnologia allo scopo di fornire credibilità ad essa ma soprattutto di chiarirne i limiti e la complessità d'implementazione.

A GUIDE TO EXPERT SYSTEMS si inserisce in questo contesto ed è stato curato con particolare riguardo nella forma di esposizione per risultatre comprensibile a chi non ha avuto precedente esperienza di AI. Si rivolge infatti ad un'utenza estremamente estesa: manager d'azienda, professionisti generici, studenti o programmatori di provenienza estranea al campo AI, ecc....

D.A. Waterman analizza in modo succinto i principali formalismi per la rappresentazione della conoscenza, i linguaggi procedurali per la manipolazione simbolica ed i tools per la costruzione di sistemi esperti con particolare riferimento alle strutture di rappresentazione della conoscenza dichiarativa o procedurale. L'analisi è necessariamente poco approfondita, dato l'obiettivo di rimanere facilmente comprensibile, ma sufficientemente dettagliata da rendere possibile una chiara differenziazione tra la programmazione tradizionale (di tipo algortimico) e la programmazione simbolica e knowledge based.

Nella seconda parte l'autore esemplifica in modo concreto le problematiche connesse alla scelta di intraprendere la costruzione di un sistema esperto e presenta un insieme di criteri che aiutano a definire il campo di applicabilità e quindi l'adeguatezza o meno di un sistema esperto in riferimento alla risoluzione di problemi specifici.

L'Autore analizza alcuni problemi che sorgono comunemente nella fase (per altro spesso la più lunga e difficile) di trasferimento di conoscenza dell'esperto del dominio al knowledge engineer e quindi di formalizzazione di questa in un linguaggio specifico. Anche qui presenta alcuni criteri da seguire. È un tentativo direndere meno "artistico" e maggiormente scientifico il lavoro del knowledge engineer. Quest'aspetto è legato ad uno dei settori di ricerca può interessanti e più giovani del campo dei sistemi esperti che è volto alla scoperta di nuovi metodi per l'acquisizione della conoscienza che possano condurre eventualmente ad automatizzare il processo: traguardo, questo, ancora relativamente distante.

Particolarmente utile sia per il profano sia per lo specialista risulta la presentazione, nella terza parte del libro, dei cataloghi dei più importanti sistemi esperti e tools costruiti negli ultimi quindici anni. Il catalogo di sistemi esperti è diviso per campi di applicazioni e diogni sistema esperto vengono descritte le principali caratteristiche: dominio di applicazione, metodo di rappresentazione della conoscenza, strategia inferenziale, linguaggio o tool d'implementazione, luogo di sviluppo, stadio di sviluppo raggiunto. Il catalogo dei tools è suddiviso per tipo di formalismo di rappresentazione. I cataloghi sono naturalmente corredati di dettagliata bibliografia anch'essa suddivisa per argomenti.

a cura di Hadley Taylor

IL NOTIZIARIO SUI CD-ROM

a cura di A. Borgia (+°), *G. Dalto* (°), *G. Ferrari* (+°), *C. Pagani* (+°), *F. Vagliasindi* (+°)

• **CAR** - La Philiphs ha investito 15.000.000 di sterline per sviluppare un sistema "computer assisted retrieval" (CAR), utilizzando il CD-ROM, per mettere a disposizione degli automobilisti tutte le informazioni necessarie per viaggiare con risparmio di tempo e "benzina".
Il sistema chiamato CARIM si compone di un computer di bordo (munito di video display, tastiera e lettore di CD), di un rilevatore della posizione del veicolo, di un sensore per lo stato delle normali funzioni del veicolo (benzina, temperatura del motore ecc. ecc.) e di un'autoradio per ricevere tutte le informazione sul traffico.
Sul dischetto vengono riprodotte le mappe stradali cittadine (in un secondo momento anche extraurbane) con un particolare sistema di memorizzazione; l'intera città di Amsterdam, per esempio viene ad occupare solo lo 0.045% della capacità di memoria del CD.
La Philiphs prevede di commercializzare il prodotto verso la fine del 1987, ad un costo che si aggirerà sulle 1.800 - 2.000 sterline.

• **BIP** - La Electronic Publishing Division del R.R. Bowker ha riprodotto su CD-ROM tutta la serie di volumi dei "Books in Print" e della "Ulrich's international periodicals directory". I due data-base si chiamano rispettivamente BIP Plus e Ulrich's Plus, hanno aggiornamento quadrimestrale e vengono prodotti in due versioni in relazione al tipo di lettore impiegato (Sony o Hitachi).

• **ERIC** - Dialog Information Service ha annunciato per la fine di giugno 1987 il data-base ERIC su CD-ROM. Esso fornisce gli abstracts di articoli tratti da circa 700 periodici specializzati nel campo dell'educazione, nonché numerose altre pubblicazioni, a partire dal 1981; avrà un aggiornamento quadrimestrale.

• **The Universe of Sound, Volume One** - Un prodotto consistente di CD-ROM disk drive, una speciale interfaccia e un disco CD-ROM, lavora con EMULATOR II, una semplice tastiera digitale fatta dall'EMU Systems Inc. di Santa Barbara. Il disco CD-ROM contiene migliaia di suoni di strumenti diversi, di effetti sonori e di combinazioni di sistemi completamente digitalizzati. Il controllo dell'accesso è fatto attraverso un Apple Macintosh oppure con una speciale interfaccia tra l'Emulatore e il Sony CD-ROM drive. Potrebbe essere felicemente utilizzato nelle sale di registrazione o nei concerti dal vivo.

• **LASERTRACK PATHFINDER** - Prodotto dalla LaserTrack Corporation contiene carte di navigazione e mappe per ricerca di piani di volo alternativi. Fornisce un servizio di informazioni completo per piloti e navigatori. La frequenza di aggiornamento prevista è di 28 giorni.

(°) *Dipartimento di Scienze dell'Informazione*
(+) *Media Group S.r.l.*

• **PC Laser Library** - Prodotto dalla Library Corporation contiene circa 9000 programmi SW di pubblico dominio per PC IBM. L'abbonamento costa 400 $ l'anno e la frequenza d'aggiornamento prevista è di 2 anni.

• **CD-ROM ATLAS** - Il primo atlante geografico su CD-ROM, prodotto da Delorme Mapping Systems of Freeport contiene confini politici, strade, città paesi, fiumi, laghi, isole, monti e profondità degli Oceani. La scala di visualizzazione su monitor è di 20 miglia ogni inch. Contiene anche informazioni del governo come il CENSUS BUREAU delle mappe delle strade.

• **LISA** - A febbraio è uscita la versione definitiva di LISA (Library and Information Science Abstracts) contiene dati completi dal 1969 al 1986.

• **TMS** - La TMS Inc. ha annuinciato la disponibilità del suo LaserDos Origination Software come complemento dell'annunciato MS-DOS CD-ROM extension. I due prodotti insieme permettono la produzione di dischi CD-ROM che sono conformi allo standard High-Sierra Group.

• **Post Office** - 23.500.000 indirizzi postali della Gran Bretagna sono stati messi su un singolo CD-ROM. Il prodotto utilizza il software di retrieval della Silver Platter e hardware Hitachi, ogni singolo indirizzo può essere localizzato in due secondi. Il prezzo dell'abbonamento per due anni comprendente il disco di aggiornamento è di 3.000 sterline.

• **Knowledgeset,** la società che ha preparato l'Enciclopedia Grolier, sta concentrando i suoi sforzi sul mercato delle associazioni professionali producendo CD-ROM contenenti documentazione tecnica. Alla conferenza tenuta in Ottobre a New York Kildall è stato molto critico verso il CD-I ed ha affermato che la prima reale applicazione CD-I non uscirà prima del 1990.

• **Chioschi** -Harold Kester della DELMAR GROUP ha già installato più di 100 punti di vendita su CD-ROM nelle librerie. Hanno registrato 350.000 transazioni nei loro chioschi e i sorveglianti hanno constatato che il 95% degli utenti trova il sistema CD-ROM facile da usare. Ogni terminale esegue 90-100 transazioni al giorno.

• **Facts On File** ha mostrato un prototipo di quello che dicono essere il primo prodotto che integra audio, grafica e testo su un solo disco. Chiamato "Visual Dictionary On CD-ROM" contiene 3.000 illustrazioni e 20.000 parole. Il prototipo è in inglese e francese e include l'audio per commenti e pronunce. Il sistema gira su APPLE II e su PC IBM e compatibili. Sarà disponibile nel 1988 in Inglese, Francese e Spagnolo.

• **Fiscovideo** - Il primo CD-ROM contenente tutte le leggi tributarie, è un prodotto nato dalla collaborazione

tra la rivista IL FISCO e la società Kronos; a breve distanza dall'annuncio del Fiscovideo, è stato annunciato un secondo prodotto contenente anch'esso leggi di carattere tributario, edito dell'editore Giuffrè in collaborazione con SIPE OPTIMATION.

• **ED.EL** - È una nuova joint-venture per la realizzazione di prodotti editoriali basati su supporti elettronici costituita pariteticamente da Espresso, Seat e Olivetti, avrà un capitale sociale di un miliardo e sarà operativa entro l'estate. Ed.El. si propone come obiettivi la gestione telematica di tutte le informazioni che richiedono frequenti aggiornamenti nonché l'utilizzo delle memorie ottiche (CD-ROM) per la "stampa elettronica" di manuali, enciclopedie, cataloghi.

CALENDARIO

• **GIUGNO 9-11** Optical Storage for Large Systems. Omni Park Central Hotel, New York, NY.

• **LUGLIO 21-23** Fifth annual Conference on the Future of Optical Memories. Hyatt on Union Square. San Francisco, CA.

• **OTTOBRE date non definite** Third International Conference on Optical Memory Technology. Keeidanren Kaikan, Tokio, Japan.
Data Show '87. Tokio International Fairgrounds, Harumi, Tokio.

• **NOVEMBRE 9-11** Third Annual Conference on Optical Storage for Small Systems. The Biltmore Hotel. Los Angeles, CA.

CALL FOR PAPERS

NINTH EUROPEAN WORKSHOP ON APPLICATIONS AND THEORY OF PETRI NETS

Wednesday 22 - Friday 24, June 1988

VENICE, ITALY

The ninth Annual European Petri Net Workshop will be organized by the *Dipartimento di Scienze dell'Informazione, Università di Milano.* Papers presenting original contributions in any area of applications and theory of Petri nets are sought. The language of the workshop is English.

TOPICS

Net based system models
Graphical representation of systems
Interpretations of nets
Analysis of structure and behaviour of nets
Relationship between nets and other models
Educational issues
Computer tools for Petri Nets

Applications of nets to:

Office automation
Flexible manufacturing systems
Programming languages
Protocols and interfaces
Hardware structures
Real-time systems
Performance evaluation
Operations research

The workshop takes place under the auspices of: AFCET SIG "Système Parallèles et Distribués" and CNRS-C^3, AICA, BCS SIG "Formal Aspects of Computing Science", EATCS and GI SIG "Petri Nets and Related System Models".

PAPERS

Authors are invited to submit 10 copies of an extended abstract (approx. 10 pages) or 10 copies of a full draft paper (at most 30 pages). If possible, authors should classify their contribution according to the topics listed above.

The workshop will also contain:

- an exibition of computer tools for Petri Nets,
- short presentations of projects where nets are put to pratical use,
- an exibition of posters describing theoretical and practical results.

For tool-presentations, project-presentations, and postr-displays a short description should be submitted (1-2 pages). It is possible to combine e.g. a tool-presentation with a poster-display.

All four kinds of submissions should be received by the chairman of the program committee no later than January 15, 1988. The authors should clearly indicate the kind(s) of submission intended. Authors will be notified of acceptance/rejection by April 1, 1988. Final papers are due by May 15, 1988. The page limit will be 20 pages for papers and 10 pages for project-presentations.

PROGRAM COMMITTE CHAIRMAIN

Dan Simpson - Department of Computing & Cybernetics - Brighton Polytechnic, Moulsecoomb, Brighton BN2 4AT - England.

ORGANIZING COMMITTEE CHAIRMAIN

Giorgio De Michelis - Dipartimento di Scienze dell'Informazione - Università di Milano - Via M. da Brescia 9, I-20123 Milano, Italy

PROGRAM COMMITTEE

M. Ajmone-Marsan, Italy
C. Andre, France
F. Feldbrugge, The Netherlands
C. Girault, France
M. Lindqvist, Finland
A. Mazurkiewicz, Poland
M. Nielsen, Denmark
R. Razouk, U.S.A.
M. Silva, Spain
C. Simone, Italy
D. Simpson, G.B. (chairman)
R. Valk, F.R.G.
K. Voss, F.R.G.

The **European Workshops on Petri nets** have place annually since 1980:

1980 Strasbourg, France
1981 Bad Honnef, Germany
1982 Varenna, Italy
1983 Toulouse, France
1984 Aarhus, Denmark
1985 Espoo, Finland
1986 Oxford, Great Britain
1987 Zaragoza, Spain

The aim of the workshops is to create a forum for discussing progress in the theory and applications of Petri Nets. Typically, the workshops have 100-150 partecipants - one third of these coming from the industry while the rest are from universities and research institutions. The workshop takes place in the last week of June.

Other major activities of the Petri net community include:

a) The **Petri Net Newsletter** is published by the Special Interest Group on Petri Nets and Related System Models of the Gesellschaft für Informatik, three times a year. The newsletter contains short research articles, announcuements of meetings, abstracts of recent publications, etc. The newsletter is edited by O. Herzog, W. Reising and R. Valk and it can be ordered from the following address:

   W. Reisig - GMD - F1-P, Schloβ Birlinghoven Postfach 1240, D-5205 St. Augustin 1 - F.R.G.

b) The **Advanced in Petri Nets** series is published by the Springer-Verlag (within the Lecture Notes in Computer Science series). These volumes are edited by G. Rozemberg. The intention of this series is to present the most significant recent results in the application and theory of Petri Nets to the broad computer science community. The majority of papers appearing in the Advances in a revised and improved form. Research papers and survey articles for the Advances can also be submitted directly to:

   G. Rozenberg - Department of matematics & Computer Science - University of Leiden, Niels Bohr Weg 1 - Postbox 9512, N-2300 RA Leiden - The Netherlands

c) The **Advanced Courses on Petri Nets** are organized periodically, in order to present the progress of Petri Nets in a systematic way. Until now there have been two such courses: Hamburg 1979 and Bad Honnef 1986. The material from these courses has also been published by Springer-Verlag (within the Lecture Notes in Computer Science series).

The activities of the Petri net community are coordinated by the following steering committee:

STEERING COMMITTEE

H.J. Genrich, Germany
K. Jensen, Denmark
C.A. Petri, Germany (honorary member)
G. Roucairol, France
G. Rozenberg, The Netherlands (chairman).

Honeywell Bull. SW 36/37 M011001 01

CIMINO MICHELE

H B ITALIA-DIREZ.GEN. MARKETING ITALIA

VIA PIRELLI,32
20124MILANO

Ufficio Distribuzione Documentazione Gestione Mailing List
Via del Parlamento, 33 - **20098 BORGOLOMBARDO (MI)**
Trattasi di trasporto di cancelleria-stampati-moduli-documenti
interni-mobili e macchine per ufficio ESONERATI DALL'OBBLIGO
DELLE BOLLE DI ACCOMPAGNAMENTO D.P.R. 6/10/78 n. 627
art. 4 n. 8 e Circolare Ministeriale n. 72 del 23/12/78 paragrafo 10